Facsímil 03

Arquitecturas y modelos

Capítulo 08

Lo que deberías saber: arquitecturas y modelos

686f6c61 · 2026

Capítulo 08: Lo que deberías saber: arquitecturas y modelos

Entrando en el tema

Este facsímil empezó con una pregunta que parece sencilla: ¿qué hay dentro de un LLM? La respuesta no era “un chat”, ni “una API”, ni “un modelo grande”. Era una cadena de decisiones técnicas: tokens, embeddings, tensores, atención, MLP, normalización, logits, sampling, adaptación, memoria, inferencia y hardware.

Este capítulo es una revisión activa. No pretende que memorices nombres. Pretende que puedas mirar una ficha de modelo, una demo de producto o una discusión sobre hardware y preguntar con calma: qué arquitectura hay debajo, qué coste tiene usarla, qué se adaptó, qué se cuantizó, qué se está midiendo y qué parte pertenece al modelo o al sistema que lo sirve.

No es el final de la historia. Es el punto donde dejamos de hablar de modelos como cajas negras y empezamos a hablar de sistemas con piezas.

---

1. Qué es un LLM: modelo, parámetros y escala

Un LLM es un modelo entrenado para predecir tokens a partir de contexto. No guarda frases como una biblioteca. Ajusta parámetros para transformar una secuencia de entrada en una distribución de probabilidad sobre el siguiente token.[1]

La idea que debes conservar es esta: los parámetros son memoria aprendida, el contexto es memoria temporal y la salida es una distribución. Si mezclas esas tres cosas, empiezan los malentendidos.

Caso cercano: una persona pega una política interna y pregunta “¿puedo aprobar este gasto?”. El modelo no “abre una carpeta” donde está la política. Procesa el texto pegado, lo mezcla con lo aprendido durante entrenamiento y produce una respuesta probable. Si falta un dato, puede sonar seguro igualmente. Por eso el contexto y la verificación no son decoración.

Vuelve al capítulo 01 si no puedes explicar la diferencia entre parámetros, contexto, token, embedding y salida.

---

2. Transformer: de texto a tensores y atención

El Transformer convirtió texto en operaciones paralelizables sobre tensores.[2] El texto se tokeniza, cada token busca su vector, se añade posición y el bloque Transformer transforma esas representaciones capa tras capa.

La intuición importante: el modelo no lee palabras como las leemos nosotros. Trabaja con matrices. Cada capa reescribe la representación de cada posición usando información de otras posiciones.

Caso cercano: cuando escribes “el banco cerró la cuenta”, el token “banco” cambia de significado según “cuenta”. No basta con un diccionario. La atención permite que las posiciones se miren entre sí para construir significado contextual.

Vuelve al capítulo 02 si no puedes dibujar el camino mínimo: texto → tokens → ids → embeddings → bloques Transformer → logits.

---

3. Q, K, V, máscara causal y softmax

La atención no es una metáfora bonita: es una operación concreta. Cada posición produce consultas Q, claves K y valores V. Las consultas se comparan con claves, se normalizan con softmax y se usan para mezclar valores.

La forma mínima de la atención escalada es:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{d_k}\right)V$$

SÍMBOLO	QUÉ SIGNIFICA	IMAGEN MENTAL
Q	Qué busca una posición.	“Necesito saber a qué se refiere esto”.
K	Qué ofrece cada posición para ser encontrada.	“Tengo información relevante para cierto tipo de pregunta”.
V	Qué contenido se mezcla si una posición importa.	“Esto es lo que aporto a la respuesta”.
d_k	Dimensión de las claves.	Escala para que softmax no se vuelva extremo demasiado pronto.

La máscara causal impide que una posición mire tokens futuros durante generación. Sin ella, entrenaríamos un modelo que ve la respuesta antes de producirla.

Caso cercano: en una frase como “Marta dejó el portátil en la mesa porque estaba cansada”, la atención ayuda a decidir si “estaba cansada” apunta a Marta o al portátil. No hay una regla universal simple: depende del contexto aprendido y de cómo cada cabeza atiende.

Vuelve al capítulo 03 si no puedes explicar por qué softmax convierte puntuaciones en pesos que suman 1.

---

4. MLP, residual, LayerNorm, logits y sampling

La atención mezcla información entre posiciones. El MLP transforma cada posición por dentro. La conexión residual mantiene una vía de continuidad. LayerNorm estabiliza escalas. Los logits convierten el último vector en candidatos de token. Sampling decide qué token sale.

El modelo no “elige una palabra” directamente. Produce logits, los pasa por softmax y obtiene una distribución:

$$p_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

SÍMBOLO	QUÉ SIGNIFICA	QUÉ CONTROLA
z_i	Logit del token candidato i.	Preferencia bruta antes de normalizar.
T	Temperatura.	Cuánto se abre o cierra el abanico de opciones.
p_i	Probabilidad final del token i.	Opción que puede muestrearse.

Caso cercano: si un asistente escribe un email formal, temperatura baja puede mantener tono estable. Si le pides diez nombres creativos para un proyecto, una temperatura algo mayor puede abrir opciones. En ambos casos, el modelo no cambia de personalidad: cambia cómo se muestrea la distribución.

Vuelve al capítulo 04 si no puedes explicar la diferencia entre logit, probabilidad, temperatura, top-k y top-p.

---

5. Arquitecturas modernas: familias, MoE, razonamiento y multimodalidad

El Transformer base no es el único diseño posible. Hay modelos encoder-only, decoder-only, encoder-decoder, MoE, multimodales, híbridos con recuperación, modelos de razonamiento con más cómputo en inferencia y arquitecturas pensadas para contexto largo. BERT popularizó encoder-only para comprensión.[3] T5 mostró la potencia de formular muchas tareas como texto-a-texto.[4]

La idea importante no es coleccionar siglas. Es entender qué compromiso cambia cada familia: memoria, velocidad, calidad, entrenabilidad, multimodalidad, coste de servir y facilidad de adaptación.

MoE añade expertos y un router: no se activa todo el modelo para cada token. Esa idea permite aumentar capacidad manteniendo coste activo menor, aunque complica entrenamiento, balanceo y serving.[5]

Caso cercano: una plataforma que atiende dudas legales, técnicas y administrativas puede beneficiarse de rutas internas distintas. Pero si el router se equivoca, no basta con tener “muchos expertos”: hay que medir qué experto se activa, cuándo y con qué resultado.

Vuelve al capítulo 05 si no puedes explicar qué gana y qué complica un modelo MoE.

---

6. Transfer learning, destilación y modelos abiertos

La mayoría de proyectos no entrena desde cero. Parte de un modelo base, lo adapta o lo rodea de herramientas. Fine-tuning cambia pesos. LoRA entrena matrices pequeñas sobre pesos congelados.[6] QLoRA permite adaptar con menos memoria combinando cuantización y LoRA.[7]

La pregunta práctica es: ¿quiero cambiar comportamiento, aportar conocimiento, reducir coste o controlar despliegue? Cada objetivo apunta a una técnica distinta.

Caso cercano: una academia quiere que un modelo corrija entregas con una rúbrica concreta. Si el modelo ya entiende el tema pero no sigue bien el formato, quizá bastan ejemplos y validación. Si necesita tono y criterios muy específicos, LoRA puede tener sentido. Si falta información cambiante, RAG será mejor que ajustar pesos.

Vuelve al capítulo 06 si no puedes distinguir fine-tuning, LoRA, QLoRA, destilación, cuantización y modelo abierto.

---

7. Inferencia optimizada, edge AI y hardware

Un modelo no se usa en abstracto. Se sirve. Y servirlo significa gestionar memoria, colas, caché, kernels, precisión, batch, latencia y coste. El capítulo anterior separó `prefill` y `decode`, que es una de las divisiones más útiles para pensar rendimiento.

Ejemplo de fórmula. Una aproximación sencilla de latencia, útil para discutir producto antes de medir con un runtime real, es:

$$T_{\text{total}} \approx T_{\text{prefill}} + n_{\text{salida}} \cdot T_{\text{decode}}$$

PIEZA	QUÉ PREGUNTA RESPONDE	POR QUÉ IMPORTA
T_{prefill}	¿Cuánto tarda en leer el contexto?	Contextos largos retrasan el primer token.
n_{salida}	¿Cuánto texto generará?	Respuestas largas encarecen decode.
T_{decode}	¿Cuánto cuesta cada token nuevo?	Generar es secuencial y depende de KV cache.

La KV cache puede ocupar varios GB con usuarios simultáneos. FlashAttention reduce movimiento de memoria en atención.[8] PagedAttention organiza mejor esa caché en serving.[9] GQA reduce memoria de claves y valores.[10]

Caso cercano: si tu asistente legal recibe contratos largos y genera respuestas cortas, el cuello puede estar en prefill. Si tu herramienta redacta informes largos desde un prompt breve, el cuello puede estar en decode. Si veinte personas lo usan a la vez, quizá el problema no es el modelo, sino la KV cache y el scheduler.

Vuelve al capítulo 07 si no puedes explicar TTFT, tokens/s, KV cache y por qué un modelo que cabe solo puede no caber con varios usuarios.

Diseccionando un LLM: de una petición al siguiente token

Hasta ahora hemos repasado el facsímil capítulo a capítulo. Ahora hagamos lo que más se usa en la vida real: seguir una petición completa. No como dibujo bonito, sino como herramienta de depuración. Cuando una respuesta sale lenta, cara, repetitiva, truncada o falsa, necesitas saber en qué capa mirar.

La inspiración visual de esta sección viene de explicaciones interactivas como *The Anatomy of an LLM*, de Roy van Rijn, que recorre tokenización, embeddings, atención, logits, sampling, post-training, KV cache y cuantización en una cadena visual.[11] Aquí lo rehacemos con la mirada del facsímil: qué cambia, qué medimos y qué decisión tomaría un equipo.

La escena: una llamada que parece sencilla

Imagina una aplicación de soporte interno. El usuario pregunta:

```
Un alumno pide ampliar el plazo de entrega porque ha tenido una incidencia técnica
documentada.
¿Qué categoría, riesgo y siguiente acción recomiendas?
```

La aplicación no manda solo esa frase. Normalmente ensambla una petición con varias capas:

CAPA DE ENTRADA	EJEMPLO	POR QUÉ IMPORTA
Instrucción de sistema	“Responde solo con JSON válido y cita la política usada.”	Define el contrato de comportamiento.
Documentos	Políticas internas sobre ampliaciones e incidencias.	Aportan conocimiento vivo que no debería inventarse.
Mensaje de usuario	La consulta concreta.	Es la tarea que cambia en cada llamada.
Contrato de salida	Campos `categoria`, `riesgo`, `accion_recomendada`, `fuente`.	Permite validar automáticamente.
Parámetros	`temperature`, `top_p`, `max_tokens`, `stop`, `seed`.	Controlan variabilidad, longitud y parada.

La primera lección práctica es esta: **el prompt real no es el texto que escribe el usuario**. Es el paquete completo que la aplicación construye. Si no registras ese paquete, luego no sabes qué ha visto el modelo.

Fase 1: texto, tokens e IDs

El modelo no recibe letras como tú las ves. Recibe IDs de tokens. El tokenizer parte el texto en piezas y asigna a cada pieza un número. Esa conversión afecta a coste, contexto, truncamiento y rarezas: código, acentos, números largos, JSON y nombres propios pueden partirse de formas poco intuitivas.

LO QUE VE UNA PERSONA	LO QUE NECESITA EL MODELO	RIESGO DE INGENIERÍA
“incidencia técnica documentada”	IDs de tokens	Puede ocupar más tokens de lo esperado.
Un JSON de salida	Tokens con llaves, comillas y campos	Si el modelo añade texto extra, el parser falla.
Un documento largo	Miles de tokens	Puede desplazar instrucciones o fragmentos relevantes.
Código o logs	Subtokens raros	Puede encarecer y romper patrones.

No hay que memorizar IDs. Hay que entender que el presupuesto empieza aquí. Si el tokenizer transforma una petición en 12 000 tokens, eso condiciona latencia, coste y KV cache antes de que el modelo “piense” nada.

Fase 2: embeddings y posición

Un ID de token por sí solo no tiene geometría. El token `15339` no está cerca de `15340` por tener un número parecido. El embedding layer busca una fila en una matriz aprendida y devuelve un vector. Ese vector es el punto de partida del token dentro del modelo.

Una forma sencilla de escribirlo es:

$$x_i = E[t_i] + p_i$$

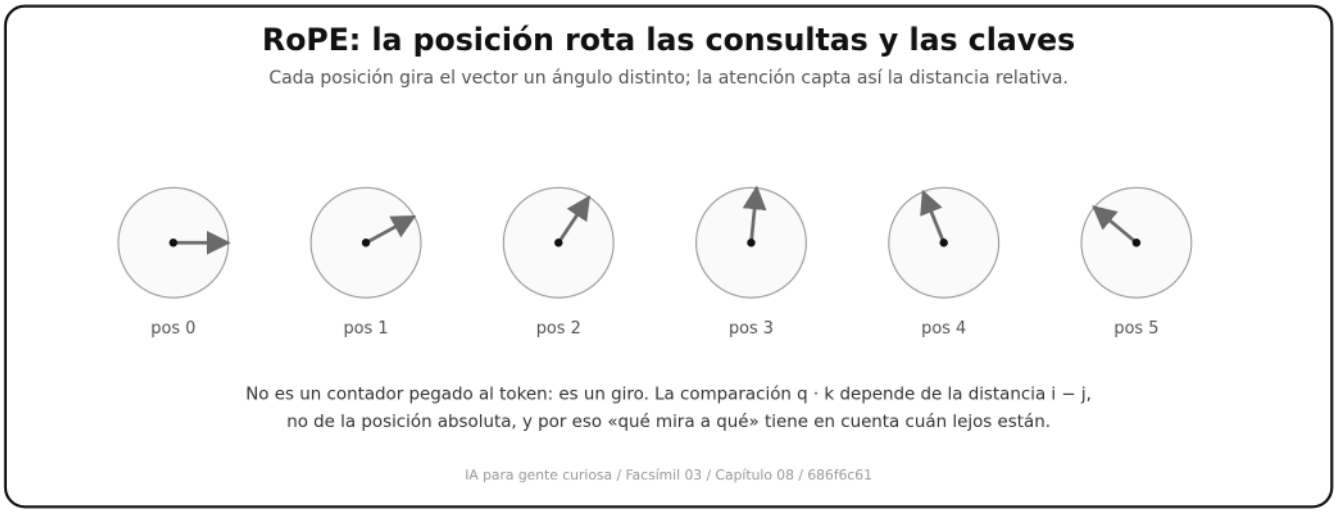
SÍMBOLO	QUÉ SIGNIFICA	EJEMPLO
t_i	ID del token en la posición i.	ID del token `incidencia`.
E	Matriz de embeddings aprendida.	Una tabla de tamaño vocabulario × dimensión.
$E[t_i]$	Fila de embedding seleccionada.	Vector de 4096 números en un modelo concreto.
p_i	Información posicional.	RoPE u otra codificación de posición.
x_i	Representación inicial del token.	Vector que entra al primer bloque.

La parte importante: el embedding inicial no es el significado final. El token “plazo” empieza con una representación, pero las capas siguientes la reescriben según “entrega”, “incidencia”, “política” y “ampliación”. La semántica útil aparece al procesar contexto.

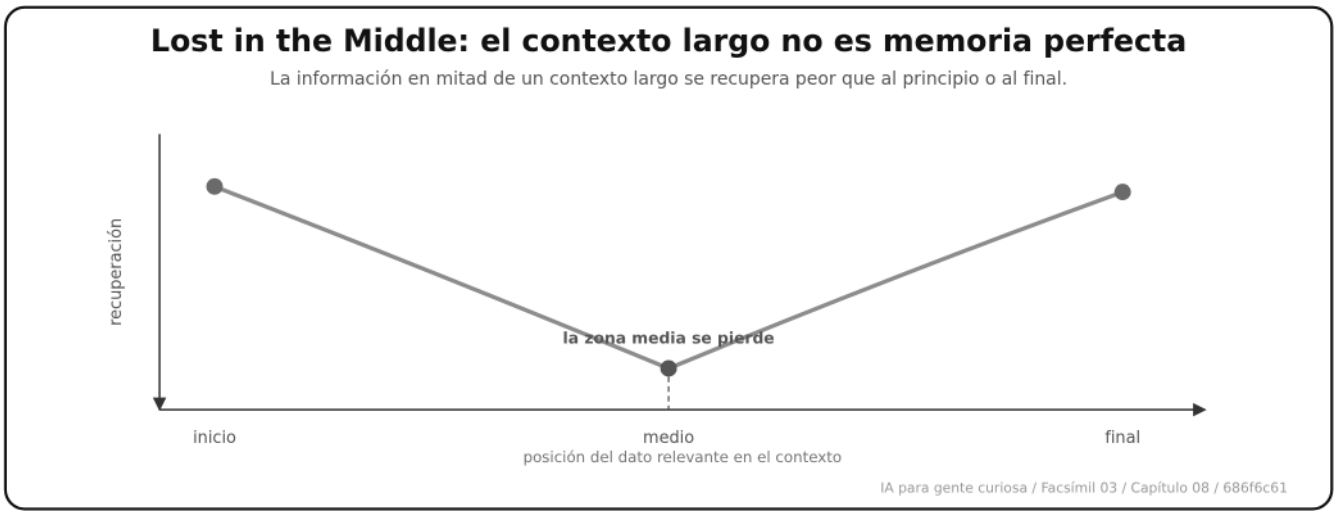
Aquí aparece un detalle que en ingeniería se nota mucho: **posición no significa “pegar un número al token y olvidarse”**. En modelos modernos es frecuente usar RoPE, que codifica posición rotando las representaciones de consulta y clave para que la atención capture relaciones relativas entre posiciones.[12] Una forma intuitiva de verlo es:

$$q_i^{\mathrm{rot}} = R(\theta_i) q_i, \qquad k_j^{\mathrm{rot}} = R(\theta_j) k_j$$

SÍMBOLO	QUÉ SIGNIFICA	POR QUÉ IMPORTA
q_i	Consulta del token en posición i.	Lo que esa posición busca en el contexto.
k_j	Clave del token en posición j.	Lo que otra posición ofrece para ser encontrada.
$R(\theta_i)$	Rotación dependiente de posición.	Mete orden sin convertir la posición en un simple contador.
$q_i^{\mathrm{rot}}(k_j^{\mathrm{rot}})^\top$	Comparación ya sensible a distancia relativa.	Ayuda a que “qué mira a qué” dependa también de dónde está.



Esto no convierte el contexto largo en memoria perfecta. El trabajo *Lost in the Middle* mostró que los modelos pueden recuperar peor información situada en zonas intermedias de contextos largos que información al principio o al final.[13] Para una aplicación RAG esto es muy práctico: no basta con “meter más documentos”. Hay que ordenar evidencias, repetir señales críticas, chunkear con criterio y medir recuperación en la posición donde realmente cae el dato.



Fase 3: bloques Transformer

Un Transformer decoder-only repite un bloque muchas veces. Cada bloque suele tener atención, normalización, conexión residual y MLP. En lenguaje de ingeniería, cada capa toma un tensor de forma aproximada:

```
[batch, secuencia, d_model]
```

y devuelve otro tensor con la misma forma, pero con representaciones más contextualizadas.

PIEZA	QUÉ HACE	CÓMO SE ROMPE EN PRODUCCIÓN
Atención causal	Permite que cada posición use tokens anteriores.	Contexto largo aumenta coste y memoria.
Q, K, V	Calculan qué busca cada token, qué ofrecen los demás y qué contenido se mezcla.	Más heads y contexto implican más trabajo.
Residual	Mantiene una vía de continuidad entre capas.	Sin ella, entrenar redes profundas sería mucho más inestable.
LayerNorm/RMSNorm	Mantiene escalas numéricas manejables.	Cambios de precisión pueden afectar estabilidad.
MLP/SwiGLU/MoE	Reescribe cada posición por dentro.	En MoE aparece routing, balanceo y coste activo.

Si un alumno se queda solo con “la atención mira palabras”, pierde la ingeniería. La atención mueve información entre posiciones; el MLP transforma la representación de cada posición; la normalización estabiliza; la residual conserva señal; y repetir esto muchas capas produce el vector final desde el que saldrán logits.

Hay tres matices que conviene dejar claros porque aparecen mucho cuando se leen implementaciones reales, model cards y artículos técnicos como la guía visual de Kato.[14]

1. Las cabezas de atención no son trozos mágicos del texto. Cada head aprende proyecciones distintas:

$$Q_h = XW_h^Q, \qquad K_h = XW_h^K, \qquad V_h = XW_h^V$$

PIEZA	QUÉ CAMBIA	LECTURA DE INGENIERÍA
X	Representaciones actuales de los tokens.	No son palabras originales; son vectores ya reescritos por capas previas.
W_h^Q	Proyección de consultas para la cabeza h.	Define qué tipo de relación puede buscar esa cabeza.
W_h^K	Proyección de claves.	Define cómo se ofrece cada posición a ser atendida.
W_h^V	Proyección de valores.	Define qué contenido se mezcla cuando una posición importa.

En producción esto importa porque `attention_heads`, `kv_heads`, `head_dim` y `d_model` no son números decorativos. Condicionan memoria, throughput, compatibilidad con kernels y tamaño de KV cache. GQA, por ejemplo, permite que varias cabezas de consulta compartan menos cabezas de claves y valores; por eso puede reducir memoria de KV cache sin eliminar todas las cabezas de atención.[15]

2. El residual stream es la autopista por la que viaja la señal. En muchos decoders modernos, una capa se parece más a esto que a un bloque lineal limpio:

$$x' = x + \mathrm{Attention}(\mathrm{RMSNorm}(x))$$

$$x_{\mathrm{next}} = x' + \mathrm{MLP}(\mathrm{RMSNorm}(x'))$$

RMSNorm normaliza por raíz cuadrática media y prescinde del centrado de LayerNorm, lo que puede simplificar y acelerar ciertos diseños.[16] La idea práctica: si cuantizas, cambias precisión o miras activaciones, no estás tocando “texto”; estás tocando el flujo numérico por

el que cada capa suma pequeñas correcciones. Por eso un modelo puede degradarse de manera rara: no siempre falla todo, a veces se vuelven frágiles ciertos formatos, idiomas, longitudes o tareas.

3. El MLP no es relleno entre atenciones. En muchos modelos, el MLP usa variantes tipo SwiGLU:

$$\mathrm{SwiGLU}(x) = \big(\mathrm{swish}(xW_{\mathrm{gate}}) \odot xW_{\mathrm{up}}\big)W_{\mathrm{down}}$$

PARTE	QUÉ HACE	EJEMPLO DE INTUICIÓN
W_{gate}	Abre o cierra rutas internas.	“Este patrón semántico sí aplica aquí”.
W_{up}	Expande dimensión intermedia.	Permite más capacidad de transformación.
$\odot$	Producto elemento a elemento.	Combina activación y contenido.
W_{down}	Devuelve al tamaño del modelo.	Reintegra la señal al residual stream.

SwiGLU y otras variantes de GLU se usan porque mejoran el bloque feed-forward frente a activaciones más simples en Transformers.[17] Esta es una buena vacuna contra una frase peligrosa: “la atención lo hace todo”. No. La atención enruta información entre posiciones; el MLP transforma cada posición; y ambas piezas se alternan capa a capa.

Fase 4: logits, softmax y política de decoding

Cuando el modelo termina de procesar el contexto, todavía no ha escrito texto. Produce logits: una puntuación por token candidato del vocabulario. Después convertimos esas puntuaciones en probabilidades.

$$P(t_i \mid contexto) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

SÍMBOLO	QUÉ SIGNIFICA	DECISIÓN PRÁCTICA
z_i	Logit del token candidato i.	Preferencia bruta antes de normalizar.
T	Temperatura.	Baja para JSON estricto; más alta para ideación.
j	Todos los tokens candidatos considerados.	En vocabularios reales pueden ser cientos de miles.
$P(t_i \mid contexto)$	Probabilidad del token i dado el contexto.	No es verdad: es probabilidad de continuación.

Después llegan filtros y penalizaciones:

PARÁMETRO	QUÉ CAMBIA	CUÁNDO TOCARLO	QUÉ MEDIR
`temperature`	Aplana o concentra la distribución.	Bajar en salidas estructuradas; subir en generación creativa.	Entropía, tasa de JSON válido, diversidad.
`top_k`	Limita a los k tokens más probables.	Evitar candidatos raros sin cerrar demasiado.	Repetición, calidad, diversidad.
`top_p`	Conserva el conjunto mínimo con probabilidad acumulada p.	Generación abierta con control de cola.	Degeneración, diversidad, factualidad.
`min_p`	Elimina tokens demasiado pequeños frente al máximo.	Runtimes locales donde quieres cortar ruido extremo.	Tokens extraños, estabilidad.
`logit_bias`	Empuja o bloquea tokens concretos.	Forzar o evitar vocabulario técnico, formatos o palabras prohibidas.	Cumplimiento de formato y efectos secundarios.
`max_tokens`	Limita longitud de salida.	Control de coste y truncamiento.	Respuestas cortadas, coste por tarea.
`stop`	Para al encontrar secuencias concretas.	Evitar texto extra tras JSON o secciones delimitadas.	Tasa de salida limpia.

Aquí conviene repetirlo porque evita mucho humo: **los parámetros de decoding no hacen al modelo saber más**. Cambian cómo se selecciona el siguiente token a partir de lo que el modelo ya ha calculado.

También puede aparecer **speculative decoding**. No cambia la arquitectura mental de "siguiente token", pero sí la ingeniería de serving: un modelo pequeño o una cabeza rápida propone varios tokens candidatos y el modelo grande verifica cuáles acepta.[18] Si los acepta, ganas velocidad; si no, vuelves al modelo grande. No es una técnica para mejorar calidad: es una técnica para reducir latencia manteniendo la distribución objetivo bajo ciertas condiciones.

PIEZA	QUÉ HACE	CUÁNDO MERECE LA PENA
Modelo draft	Propone tokens baratos.	Cuando sus propuestas suelen coincidir con el modelo grande.
Modelo target	Verifica y corrige.	Cuando no quieres cambiar la calidad del modelo principal.
Ratio de aceptación	Porcentaje de tokens draft aceptados.	Si es bajo, el mecanismo añade complejidad sin acelerar.
Métrica útil	Tokens/s y p95 de latencia.	El objetivo es rendimiento, no “razonamiento mejor”.

Fase 5: prefill, decode y KV cache

La generación tiene dos fases operativas muy distintas:

FASE	QUÉ OCURRE	MÉTRICA TÍPICA	PROBLEMA TÍPICO
Prefill	El modelo procesa todo el contexto inicial.	TTFT, tiempo hasta primer token.	Contextos largos, documentos pegados, RAG excesivo.
Decode	El modelo genera token a token.	tokens/s, tiempo total, coste de salida.	Respuestas largas, batch, GPU saturada.

Durante decode, el modelo no quiere recalcular todo el contexto una y otra vez. Guarda claves y valores de atención ya calculados: la KV cache. Esa caché no es “memoria del usuario” ni “recuerdo semántico”. Es memoria de cálculo para acelerar generación autoregresiva.

Ejemplo de fórmula. Para estimar orden de magnitud de KV cache:

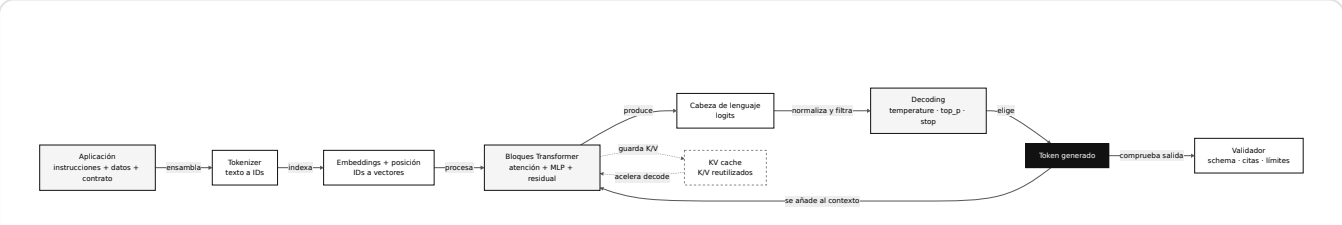
$$M_{\mathrm{KV}} \approx 2 \cdot L \cdot B \cdot S \cdot H_{\mathrm{KV}} \cdot d_h \cdot \mathrm{bytes}$$

SÍMBOLO	QUÉ SIGNIFICA	EJEMPLO
2	Guardamos claves y valores.	K y V.
L	Número de capas.	32.
B	Batch o secuencias activas.	4 usuarios simultáneos.
S	Contexto reservado o usado.	4096 tokens.
H_{KV}	Cabezas KV.	8 si hay GQA.
d_h	Dimensión por cabeza.	128.
bytes	Bytes por valor.	2 en FP16/BF16.

Esta fórmula no sustituye a medir en vLLM, SGLang, llama.cpp, Ollama o el proveedor que uses. Sirve para no prometer imposibles. Si duplicas contexto o batch, la KV cache crece. Si el modelo “cabe” sin usuarios, quizá no cabe con veinte sesiones largas.

Fase 6: el token sale y el bucle vuelve a empezar

El token elegido se añade a la secuencia. Si hay streaming, el usuario puede verlo enseguida. Si hay salida estructurada, tu aplicación no debería confiar todavía: debe parsear, validar schema, comprobar campos obligatorios y decidir si acepta, reintenta, corrige o escala.



Si algo falla, dónde miro primero

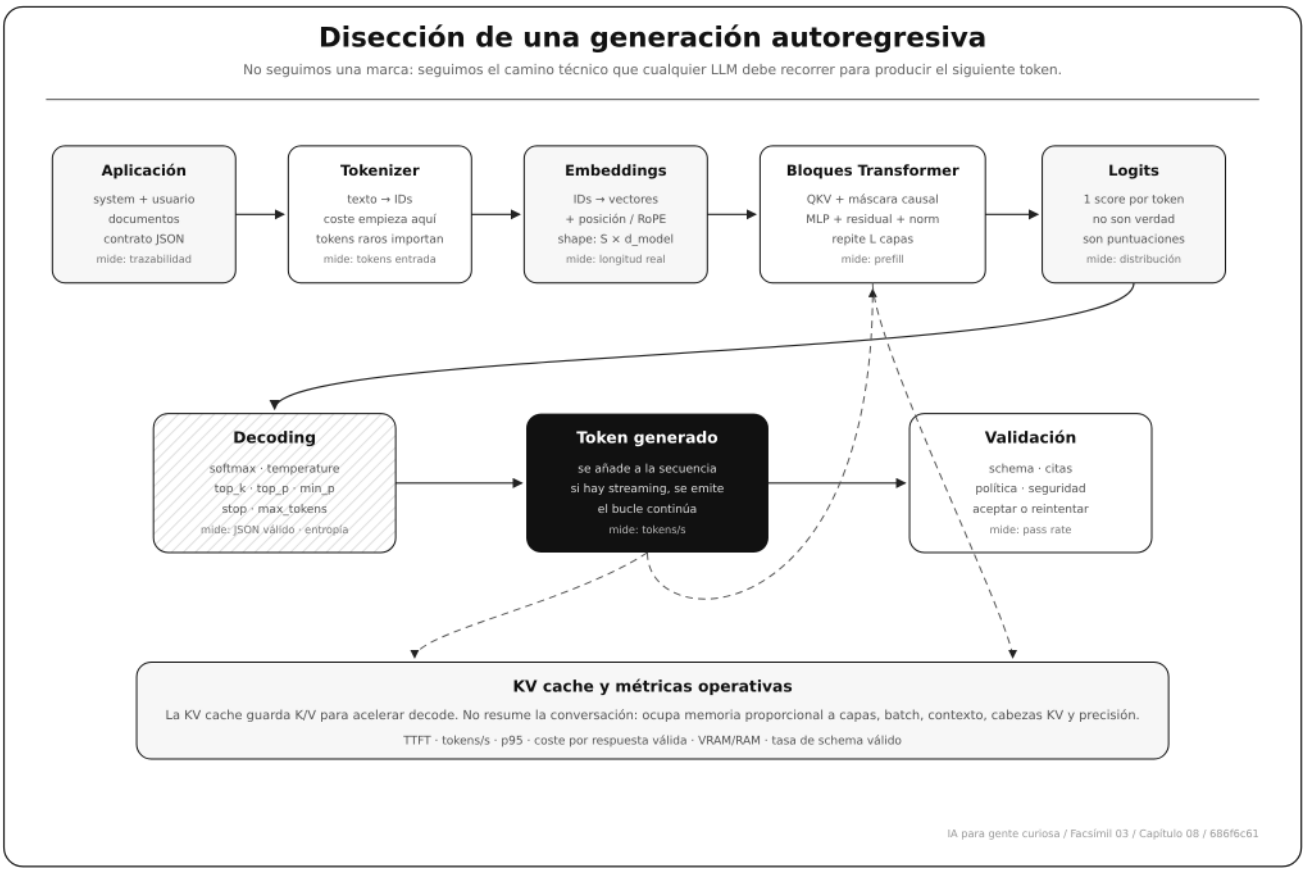
Esta es la parte más útil para ingeniería. La disección no sirve para recitar arquitectura; sirve para depurar.

SÍNTOMA	PRIMERA CAPA QUE MIRARÍA	QUÉ COMPROBARÍA
Respuesta truncada	Contrato de salida	`max_tokens`, `stop`, timeout, longitud esperada.
JSON inválido	Decoding y contrato	Temperatura, salida estructurada, schema, texto extra.
Tarda mucho en empezar	Prefill	Tokens de entrada, documentos pegados, RAG demasiado amplio.
Empieza rápido pero tarda mucho en terminar	Decode	Tokens de salida, tokens/s, batch, GPU, streaming.
Se queda sin memoria	Runtime	Pesos + KV cache + batch + contexto + margen.
Repite frases	Sampling	`repeat_penalty`, `frequency_penalty`, temperatura, prompt.
Responde sin citar	Contexto y validación	Retrieval, contrato de salida, groundedness, abstención.
Parece seguro pero se equivoca	Evaluación	Dataset propio, slices, referencias, revisión humana.
Cambia demasiado entre ejecuciones	Reproducibilidad	`seed`, temperatura, modelo exacto, proveedor, cache, herramientas.
Usa mal una herramienta	Agente/tooling	Schema, permisos, argumentos, trazas, validadores.

Una frase que debería quedarse: **la salida visible es el final de una cadena, no el lugar donde empieza la explicación**. Si el sistema falla, no preguntes solo “qué ha dicho el modelo”; pregunta qué texto entró, cuántos tokens ocupó, qué contexto recibió, qué logits se filtraron, qué parámetros estaban activos, qué memoria se reservó y qué validador aceptó la salida.

Corte anatómico de una generación

El siguiente esquema resume la disección. No pretende mostrar todos los detalles de un modelo real; pretende poner en una misma mesa las piezas que más se tocan cuando se construye una aplicación: entrada, tensores, bloques, logits, decoding, KV cache, salida y métricas.



Qué debería llevarse un ingeniero

Después de esta disección, un ingeniero no debería decir “el modelo ha fallado” como si fuera una explicación suficiente. Debería poder formular hipótesis:

- si falla el formato, miro contrato, decoding y validador;
- si falla la evidencia, miro retrieval, contexto y citas;
- si falla la latencia inicial, miro prefill y tokens de entrada;
- si falla la latencia total, miro decode y longitud de salida;
- si falla memoria, miro pesos, KV cache, batch, contexto y precisión;
- si falla estabilidad, miro parámetros, versión de modelo, seed, proveedor y evaluación repetida.

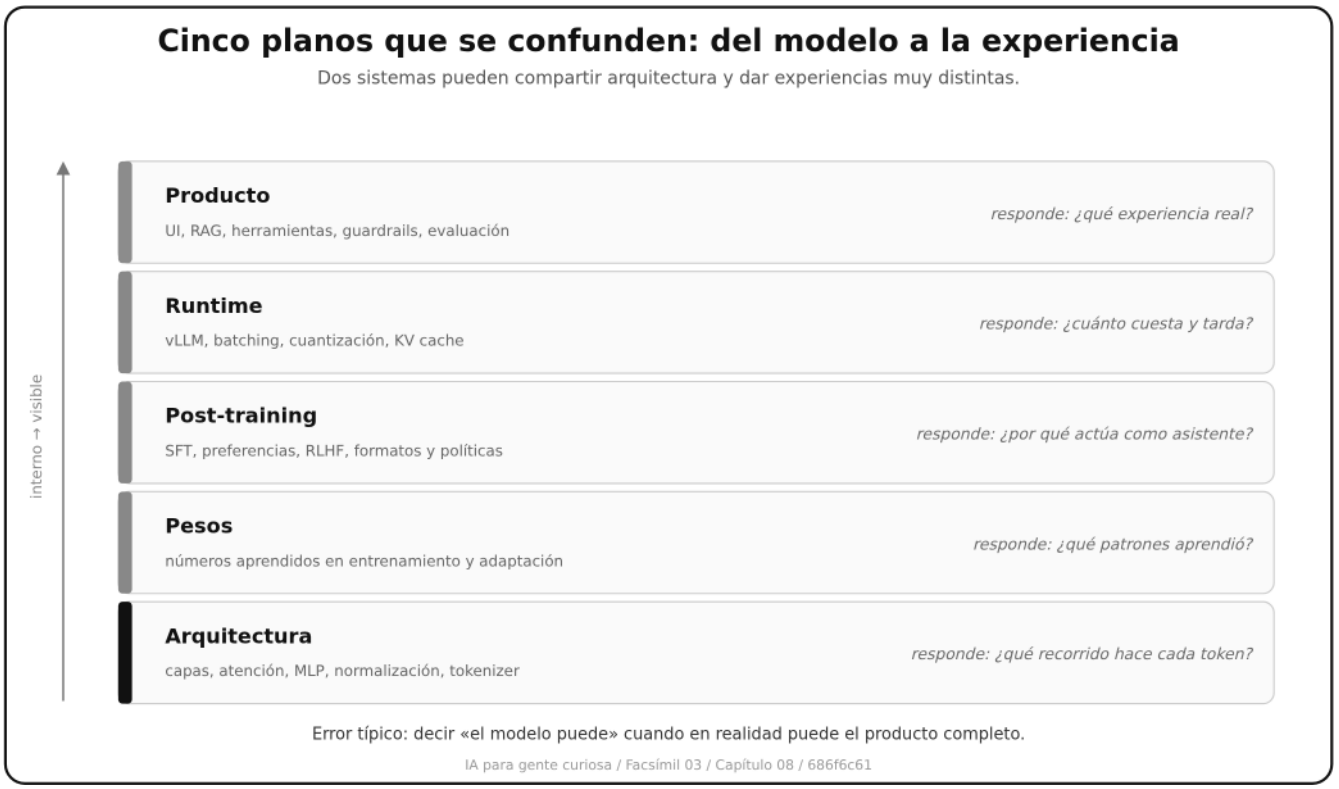
Esa mirada es la diferencia entre usar LLMs como una caja negra y trabajar con ellos como sistemas de ingeniería.

Arquitectura, pesos, post-training y producto no son lo mismo

Una de las ideas más útiles del artículo de Kato es separar piezas que en conversación se mezclan demasiado. Dos modelos pueden compartir “familia Transformer” y comportarse de forma muy distinta; dos despliegues pueden servir pesos parecidos y dar experiencias

distintas; una API puede esconder optimizaciones que no están en el paper del modelo.

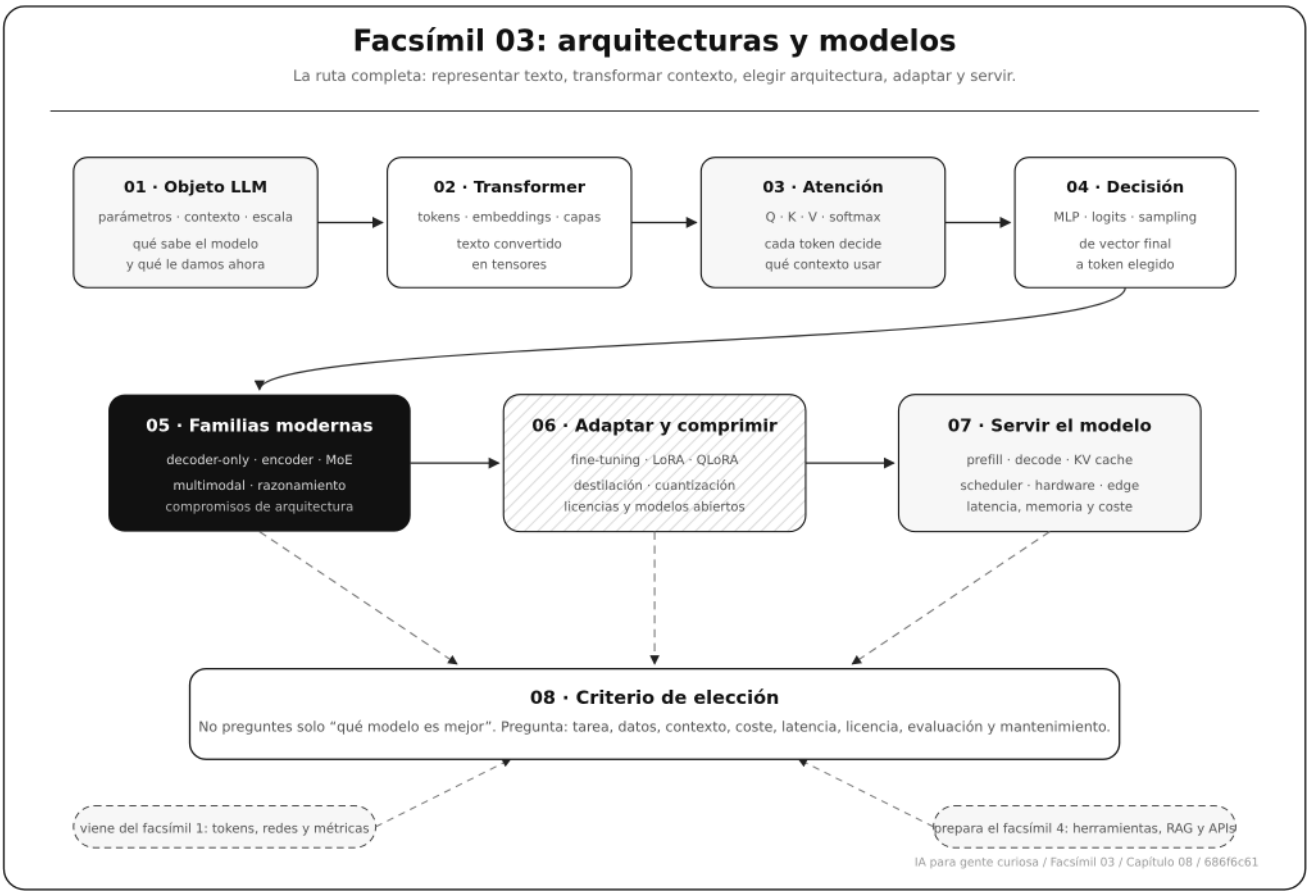
PLANO	QUÉ ES	QUÉ PREGUNTA RESPONDE	ERROR TÍPICO
Arquitectura	Forma del modelo: capas, atención, MLP, normalización, tokenizer, contexto, MoE o no MoE.	¿Qué recorrido computacional hace cada token?	Creer que “Transformer” ya explica el comportamiento final.
Pesos	Números aprendidos durante entrenamiento y adaptación.	¿Qué patrones ha aprendido el modelo?	Llamar abierto a un modelo solo porque hay una demo pública.
Post-training	SFT, preferencias, RLHF/RLAIF/RLVR, herramientas, formatos y políticas.	¿Por qué responde como asistente y no como modelo base?	Atribuir al pretraining cosas que vienen de alineamiento o instrucciones.
Runtime	vLLM, SGLang, llama.cpp, servidor propietario, batching, cache, cuantización.	¿Cuánto cuesta, cuánto tarda y cuántos usuarios aguanta?	Comparar modelos sin comparar configuración de serving.
Producto	UI, memoria externa, RAG, herramientas, guardrails, evaluación, permisos.	¿Qué experiencia real tiene la persona?	Decir “el modelo puede” cuando realmente puede el sistema completo.



Esta separación ayuda mucho cuando se comparan modelos cerrados, pesos abiertos y código abierto. Un modelo con pesos descargables puede no tener datos de entrenamiento abiertos. Un modelo con licencia permisiva puede exigir un runtime concreto para rendir bien. Un proveedor cerrado puede ofrecer gran calidad y tooling, pero menos control sobre pesos, tokenizer, kernels o cambios de versión. La decisión profesional no es ideológica: es trazabilidad, coste, privacidad, capacidad de depuración, licencia, evaluación y riesgo operativo.

El mapa completo del facsímil

El facsímil tiene una forma: empieza en el objeto “LLM”, entra en sus capas internas, abre el abanico de arquitecturas, baja a adaptación y termina en el coste real de usarlo. Este mapa intenta mostrar esa escalera.



En el día a día

Este facsímil aparece cada vez que alguien dice “usemos IA” y todavía no ha dicho qué modelo, para qué tarea, con qué contexto, con qué coste y bajo qué límites.

En producto, te ayuda a no elegir por fama. Un modelo enorme puede ser peor decisión si tu caso necesita baja latencia, ejecución local o una licencia concreta. Un modelo más pequeño puede ser suficiente si el dominio está bien acotado, hay recuperación externa y la evaluación es clara.

En ingeniería, te ayuda a separar problemas. Si la salida es mala, puede ser arquitectura, prompt, datos, decoding, herramienta, evaluación o expectativa. Si la latencia es mala, puede ser prefill, decode, KV cache, batch, hardware o red. Poner nombre a la pieza ahorra semanas de confusión.

En aprendizaje, te da una brújula: ya no estás leyendo “modelos” como si fueran marcas. Estás leyendo decisiones de diseño.

Por qué debería importarte

Porque el modelo que eliges condiciona todo lo demás: qué datos necesitas, cómo integras herramientas, cuánto cuesta servir, qué privacidad puedes prometer, qué evals debes construir y qué experiencia tendrá la persona que lo use.

También porque la mayoría de errores caros no nacen de no saber una sigla. Nacen de mezclar planos: pedirle a fine-tuning que actualice conocimiento cambiante, creer que cuantizar no afecta calidad, evaluar solo una demo, ignorar la KV cache o llamar “open source” a cualquier descarga de pesos.

Dónde solía tropezar yo

Estos tropiezos son útiles porque suenan razonables. Precisamente por eso conviene nombrarlos.

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Comparar modelos solo por tamaño	Los parámetros no dicen latencia, contexto útil, licencia, coste ni calidad en tu tarea.	Comparar con una tabla de criterios y pruebas propias.
Creer que Transformer equivale a chat	Transformer es arquitectura; el chat es una interfaz y un post-training.	Separar modelo base, modelo instruct y producto.
Meter conocimiento cambiante en fine-tuning	Ajustar pesos no es buena forma de actualizar datos vivos cada semana.	Usar recuperación o herramientas cuando el dato cambia.
Confundir cuantización con pérdida gratis	Menos bits pueden cambiar calidad, estabilidad o compatibilidad.	Medir antes/después con casos reales.
Ignorar serving	Una demo local no revela colas, KV cache, p95 ni coste por usuario.	Probar concurrencia, contexto largo y respuestas largas.
Depurar solo mirando la respuesta final	La salida puede fallar por tokenización, contexto, decoding, runtime, retrieval, schema o validador.	Diseccionar la llamada: entrada, tokens, logits, parámetros, KV cache y validación.
Recapitular como lista de siglas	Saber decir QKV, MoE o LoRA no significa entender el sistema.	Reexplicar cada pieza con entrada, salida, coste y ejemplo.

Vamos a programarlo

Ahora que hemos mirado una llamada por dentro, toca tomar una decisión de arquitectura. Esta segunda práctica convierte el criterio del facsímil en una matriz ponderada.

La práctica real está en **kit descargable**. No decide por ti, pero obliga a escribir qué pesa más: calidad, latencia, coste, privacidad, mantenimiento, adaptación y contexto externo.

ARCHIVO	QUÉ CONTIENE
`Makefile`	Atajos para ejecutar, probar y limpiar el kit.
`requirements.txt`	Declara que el ejercicio usa solo la biblioteca estándar de Python.
`data/model_strategy_case.json`	Opciones, criterios y pesos.
`contracts/model_strategy_policy.json`	Mínimos de privacidad/contexto y opción esperada.
`ops/score_model_strategy.py`	Scoring ponderado y razones de decisión.
`tests/test_lab_contract.py`	Tests que protegen las ideas del capítulo.
`templates/entrega.md`	Plantilla de entrega del alumno.
`output/model_strategy_report.json`	Puntuaciones estructuradas.
`output/model_strategy_decision.md`	Memo de decisión.

Ejecuta:

```
# Descomprime el ZIP del capítulo y ejecuta estos comandos dentro de esa carpeta
make run
cat output/model_strategy_decision.md
```

Como gate:

```
make test
```

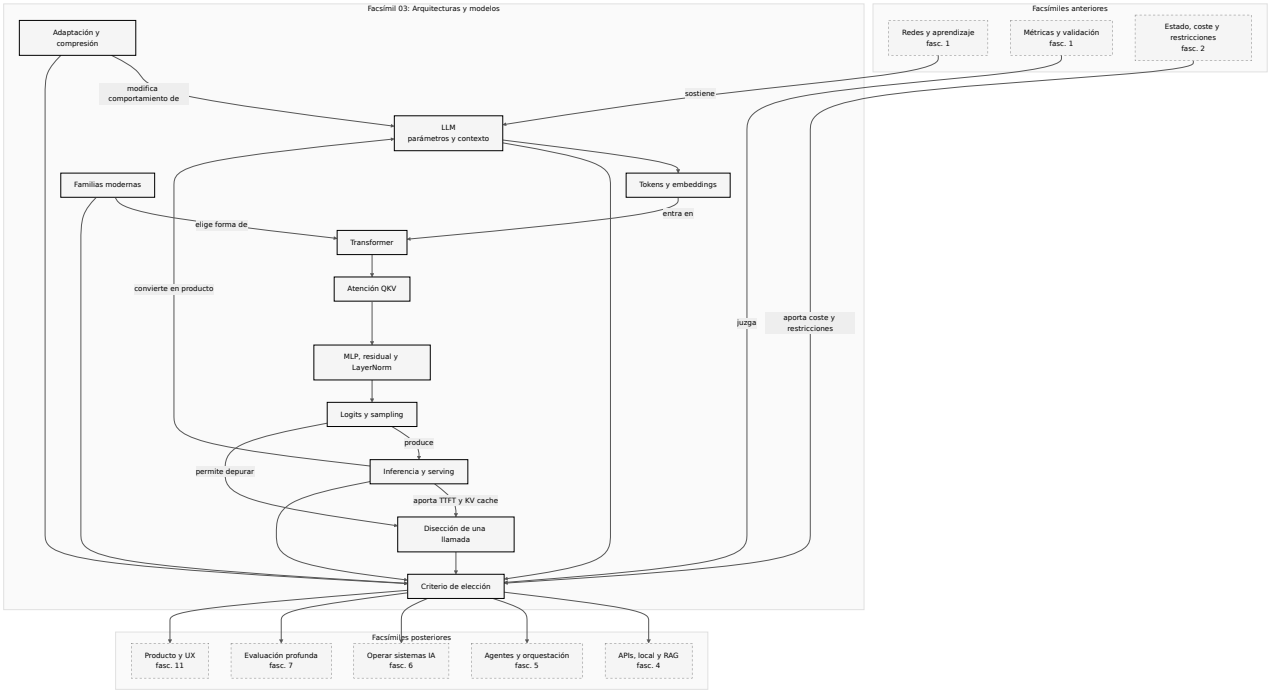
Qué deberías ver. Con el caso de partida, el memo declara un ganador (`api_con_rag`) y una tabla de puntuaciones muy ajustada: `api_con_rag` 3,75, `modelo_abierto_gpu` 3,65, `api_potente` 3,6 y `modelo_local_cuantizado` 3,5. La diferencia pequeña es el mensaje: la matriz no decide sola, obliga a escribir qué pesa más y deja una decisión revisable.

Qué entregaría un alumno. El Markdown generado, una opción nueva, pesos ajustados a su caso y una decisión defendible. Un caso más completo extendería esta misma decisión un paso más allá.

Cómo encaja todo

Este mapa no es un índice bonito: es la lectura operativa del facsímil. La primera fila responde a cómo un texto entra en un modelo y termina como token elegido. La segunda fila responde a qué decisiones aparecen cuando ese modelo deja de ser un dibujo y se convierte en una pieza que hay que adaptar, servir y mantener.

También sirve de puente. Lo aprendido en el facsímil 1 explica por qué existen pérdida, métricas y aprendizaje; el facsímil 2 enseña a pensar en estados, coste y restricciones; y el facsímil 4 tomará estas piezas para construir herramientas, RAG, APIs y sistemas locales.



Vocabulario aprendido

Antes de cerrar, conviene comprobar que estas palabras ya no son ruido. No hace falta recitarlas perfectas; hace falta poder usarlas en una conversación técnica sin perder el hilo.

TÉRMINO	DEFINICIÓN
LLM	Modelo de lenguaje grande entrenado para predecir tokens desde contexto.
Parámetro	Número aprendido durante entrenamiento.
Contexto	Tokens disponibles en una petición concreta.
Embedding	Vector que representa un token o fragmento de texto.
Transformer	Arquitectura basada en atención y bloques paralelizables.
Atención	Mecanismo que mezcla información entre posiciones.
Máscara causal	Restricción que evita mirar tokens futuros durante generación.
MLP	Subred que transforma cada posición dentro del bloque.
Residual	Conexión que conserva información entre capas.
LayerNorm	Normalización que estabiliza activaciones.
Logit	Puntuación previa a convertir candidatos en probabilidades.
Sampling	Elección del siguiente token desde una distribución.
MoE	Arquitectura con expertos y router que activa solo una parte del modelo.
Multimodalidad	Capacidad de trabajar con texto, imagen, audio u otros datos convertidos a representaciones compatibles.
LoRA	Técnica de adaptación con matrices de bajo rango.
QLoRA	LoRA sobre modelo cuantizado para reducir memoria.
Destilación	Entrenar un modelo pequeño para imitar uno mayor.
Cuantización	Representar números con menos bits para ahorrar memoria y coste.
KV cache	Memoria de claves y valores usada durante inferencia.
Serving	Sistema que ejecuta modelos para usuarios reales.
Disección de LLM	Recorrido técnico de una petición desde texto y tokens hasta logits, sampling, KV cache y salida validada.
Prefill	Fase en la que el modelo procesa todo el contexto inicial antes de emitir el primer token.

TÉRMINO	DEFINICIÓN
Decode	Fase autoregresiva en la que el modelo genera un token, lo añade al contexto y repite.
TTFT	Tiempo hasta el primer token visible para el usuario.
Entropía de salida	Medida de dispersión de la distribución de candidatos; ayuda a comparar estabilidad y diversidad.
RoPE	Codificación posicional rotatoria que incorpora orden en consultas y claves de atención.
GQA	Atención agrupada donde varias cabezas de consulta comparten menos cabezas de claves y valores para reducir KV cache.
RMSNorm	Normalización basada en raíz cuadrática media usada en muchos decoders modernos.
Residual stream	Flujo principal de activaciones al que cada subcapa suma correcciones.
Speculative decoding	Técnica de inferencia donde un modelo rápido propone tokens y el modelo principal los verifica.

Antes de pasar página

Responde sin mirar los capítulos. Si dudas, el enlace te dice dónde volver.

- [ ] ¿Puedo explicar qué diferencia hay entre parámetros, contexto y salida? (Vuelve al capítulo 01.)
- [ ] ¿Sé recorrer el camino texto → tokens → embeddings → Transformer → logits? (Vuelve al capítulo 02.)
- [ ] ¿Puedo explicar Q, K, V y softmax sin decir solo “atención”? (Vuelve al capítulo 03.)
- [ ] ¿Entiendo cómo temperatura, top-k y top-p cambian una respuesta? (Vuelve al capítulo 04.)
- [ ] ¿Puedo distinguir encoder-only, decoder-only, encoder-decoder y MoE? (Vuelve al capítulo 05.)
- [ ] ¿Sé cuándo elegir RAG, fine-tuning, LoRA, QLoRA o cuantización? (Vuelve al capítulo 06.)
- [ ] ¿Puedo separar prefill, decode, TTFT, tokens/s y KV cache? (Vuelve al capítulo 07.)
- [ ] ¿Puedo diseccionar una petición completa y decir dónde miraría si falla formato, latencia, memoria o evidencia? (Si no, vuelve a «Diseccionando un LLM: de una petición al siguiente token».)

- [ ] ¿Puedo justificar una elección de modelo con criterios, no con una marca? (Si no, vuelve a «El mapa completo del facsímil».)
- [ ] ¿Sé qué parámetro tocar para ver qué métrica se mueve al diseccionar un LLM? (Si no, vuelve a «Diseccionando un LLM: de una petición al siguiente token».)
- [ ] ¿Sé construir una matriz de decisión y ver cómo cambia el ganador al ajustar los pesos? (Si no, vuelve a «El mapa completo del facsímil».)

En resumen

La versión corta del facsímil no es “los LLMs son Transformers”. Es más concreta: un modelo es una arquitectura entrenada, adaptada y servida bajo restricciones.

IDEA FUERZA	DETALLE
Un LLM no es una API.	Es una arquitectura con parámetros, contexto, tokens, atención y sampling.
La atención no es una caja negra.	Q, K, V y softmax explican cómo una posición usa otras posiciones.
Generar texto es decidir tokens.	Logits, temperatura, top-k y top-p controlan el abanico de salida.
La arquitectura es compromiso.	MoE, multimodalidad, contexto largo y modelos híbridos cambian coste y capacidades.
Adaptar no siempre es entrenar.	RAG, fine-tuning, LoRA, QLoRA, destilación y cuantización resuelven problemas distintos.
Servir un modelo es ingeniería.	Prefill, decode, KV cache, hardware y p95 importan tanto como la calidad media.
Diseccionar una llamada ayuda a depurar.	Formato, latencia, memoria, evidencia y coste suelen fallar en capas distintas.
Elegir modelo es explicitar restricciones.	Tarea, datos, coste, privacidad, latencia, licencia, evaluación y mantenimiento.

Recursos para seguir: leer, construir y experimentar

Ya sabes qué hay dentro de un modelo de lenguaje: tokens, atención, capas, sampling y la familia de arquitecturas. Para que no se quede en teoría, aquí tienes recursos reales.

Para experimentar sin código. Lo has hecho en las cajas «Pruébalo en 5 minutos»: el visualizador de Brendan Bycroft (`bbycroft.net/llm`) para recorrer un transformer real en 3D y ver la atención por dentro; los playgrounds (`platform.openai.com` , `aistudio.google.com` , `console.anthropic.com`) para jugar con la temperatura y el top-p; Hugging Face (`huggingface.co`) para comparar modelos abiertos y leer sus fichas; y Ollama (`ollama.com`) para correr un modelo cuantizado en tu propio portátil.

Para construir. La librería de referencia para trabajar con estos modelos es Transformers de Hugging Face, sobre PyTorch; para inferencia local, Ollama y llama.cpp; para servir a escala, proyectos como vLLM. Casi todo es código abierto.

Para leer. El artículo fundacional es *Attention is All You Need* (Vaswani y otros, 2017), y para entenderlo paso a paso, *The Illustrated Transformer* de Jay Alammar es una de las mejores explicaciones visuales que existen. Cada capítulo cierra con su «Para saber más». Los cuadernos del facsímil son el puente entre el diagrama y el código.

Para saber más

Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. https://arxiv.org/abs/2305.13245

Brown, T. B. y otros (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems 33*, 1877-1901. https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

Dao, T. y otros (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems 35*. https://arxiv.org/abs/2205.14135

Dettmers, T. y otros (2023). QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems 36*. https://arxiv.org/abs/2305.14314

Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171-4186. https://doi.org/10.18653/v1/N19-1423

Fedus, W., Zoph, B. y Shazeer, N. (2022). Switch Transformers: scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120), 1-39. https://www.jmlr.org/papers/v23/21-0998.html

Hu, E. J. y otros (2022). LoRA: Low-rank adaptation of large language models. *International Conference on Learning Representations*. https://arxiv.org/abs/2106.09685

Kwon, W. y otros (2023). Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*. https://arxiv.org/abs/2309.06180

Kato. (2026). *How LLMs actually work*. https://www.0xkato.xyz/how-llms-actually-work/

Leviathan, Y., Kalman, M. y Matias, Y. (2023). Fast Inference from Transformers via Speculative Decoding. *Proceedings of the 40th International Conference on Machine Learning*. https://arxiv.org/abs/2211.17192

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. y Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics, 12*, 157-173. https://doi.org/10.1162/tacl_a_00638

Raffel, C. y otros (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *Journal of Machine Learning Research*, 21(140), 1-67. https://www.jmlr.org/papers/v21/20-074.html

Shazeer, N. (2020). GLU Variants Improve Transformer. https://arxiv.org/abs/2002.05202

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. y Liu, Y. (2024). RoFormer: Enhanced Transformer with Rotary Position Embedding. *Neurocomputing, 568*, 127063. https://doi.org/10.1016/j.neucom.2023.127063

Van Rijn, R. (2026). *The Anatomy of an LLM*. https://www.royvanrijn.com/anatomy-of-an-llm/

Vaswani, A. y otros (2017). Attention is all you need. *Advances in Neural Information Processing Systems 30*, 5998-6008. https://papers.nips.cc/paper/7181-attention-is-all-you-need

Zhang, B. y Sennrich, R. (2019). Root Mean Square Layer Normalization. *Advances in Neural Information Processing Systems 32*. https://arxiv.org/abs/1910.07467

Cuadernos para practicar

Has razonado sobre el modelo a lo largo del facsímil; estos cuadernos te dejan abrirlo y tocarlo. Son *notebooks* que se abren en Google Colab —gratis, en el navegador— o te puedes descargar. Cada uno, explicado paso a paso, con salidas reales, y anclado al capítulo del que sale.

Atención a mano: Q, K, V y la máscara causal

Qué practicas: calcular con NumPy el mecanismo de atención que mueve a todos los LLM. **Dónde encaja:** capítulos 2 y 3 (el Transformer por dentro; *queries*, *keys*, *values* y máscara causal). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Coges una frase de cinco palabras y calculas, a mano, cómo cada una «mira» a las demás: el parecido entre su pregunta (*query*) y las claves (*keys*), normalizado con *softmax*, da un mapa de atención que imprimes y dibujas. Luego aplicas la **máscara**

causal —poner menos infinito en el futuro— y ves el mapa volverse triangular: ninguna palabra puede mirar a las que vienen después, que es lo que distingue a un modelo que genera de uno que solo entiende. Terminas mezclando los valores (*values*) según esos pesos, que es justo lo que sale de la capa.

«Atención» deja de ser un eslogan: es un reparto de pesos que puedes ver casilla a casilla.

► **Abrir en Google Colab** ↓ **Descargar el cuaderno (.ipynb)**

Cómo elige palabras un LLM: temperatura, top-k y top-p

Qué practicas: controlar la creatividad de un modelo moldeando su distribución de salida. **Dónde encaja:** capítulo 4 (MLP, residual, *logits* y *sampling*). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Te pones en la última capa de un LLM: el modelo no «elige» una palabra, produce una puntuación para cada una del vocabulario y un proceso de muestreo decide. Juegas con los tres botones. La temperatura baja (0.5) agranda las diferencias y el modelo se vuelve seguro y repetitivo; la alta (1.5) las aplana y aparecen palabras improbables —creatividad y disparates—. El *top-k* se queda con las mejores; el *top-p*, con las que acumulan un porcentaje de probabilidad, adaptándose a si el modelo lo tiene claro o duda. Muestreas mil veces con cada estrategia y ves la diferencia entre un loro predecible y un poeta.

No hay un valor «correcto»: depende de si quieres un notario o un poeta. Y ese botón es tuyo, no del modelo.

► **Abrir en Google Colab** ↓ **Descargar el cuaderno (.ipynb)**

Cuantización: cómo cabe un modelo gigante en tu portátil

Qué practicas: guardar los pesos de un modelo con 8 bits y medir qué ganas y qué pierdes. **Dónde encaja:** capítulos 6 y 7 (modelos abiertos; inferencia optimizada, *edge AI* y hardware). **Qué necesitas:** un navegador. Corre en CPU; sin nada más.

Tomas una matriz de pesos en coma flotante de 32 bits y la guardas en enteros de 8 bits: buscas el peso de mayor magnitud, partes el rango en 256 escalones y redondeas cada peso al más cercano. El trato tiene dos caras y mides las dos: la memoria cae **4**

veces y el error que introduces es minúsculo —alrededor del **1%** al multiplicar por un vector—. Compruebas que con valores atípicos o con menos bits el error se dispara, que es exactamente el filo en el que viven los formatos `Q4` de los modelos GGUF.

No es magia: es redondear con cabeza. Y es la razón de que hoy corras modelos potentes en hardware modesto.

▶ Abrir en Google Colab **↓ Descargar el cuaderno (.ipynb)**

Notas

1. Brown, T. B. y otros (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems 33*, 1877-1901. https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html. GPT-3 popularizó la escala como variable central para capacidades emergentes de uso general. ↩

2. Vaswani, A. y otros (2017). Attention is all you need. *Advances in Neural Information Processing Systems 30*, 5998-6008. https://papers.nips.cc/paper/7181-attention-is-all-you-need. El paper introdujo una arquitectura basada en atención que evitaba recurrencia y facilitaba paralelismo. ↩

3. Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171-4186. https://doi.org/10.18653/v1/N19-1423. ↩

4. Raffel, C. y otros (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *Journal of Machine Learning Research*, 21(140), 1-67. https://www.jmlr.org/papers/v21/20-074.html. ↩

5. Fedus, W., Zoph, B. y Shazeer, N. (2022). Switch Transformers: scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120), 1-39. https://www.jmlr.org/papers/v23/21-0998.html. ↩

6. Hu, E. J. y otros (2022). LoRA: Low-rank adaptation of large language models. *International Conference on Learning Representations*. https://arxiv.org/abs/2106.09685. ↩

7. Dettmers, T. y otros (2023). QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems 36*. https://arxiv.org/abs/2305.14314. ↩

8. Dao, T. y otros (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems 35*. https://arxiv.org/abs/2205.14135. ↩

9. Kwon, W. y otros (2023). Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*. https://arxiv.org/abs/2309.06180. ↩

.0. Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. https://arxiv.org/abs/2305.13245. ↩

.1. Van Rijn, R. (2026). *The Anatomy of an LLM*. https://www.royvanrijn.com/anatomy-of-an-llm/. Consultado el 14 de junio de 2026. El recurso se declara en progreso; aquí se usa como inspiración pedagógica, no como fuente única ni como especificación técnica. ↩

.2. Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. y Liu, Y. (2024). RoFormer: Enhanced Transformer with Rotary Position Embedding. *Neurocomputing, 568*, 127063. https://doi.org/10.1016/j.neucom.2023.127063 ↩

.3. Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. y Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics, 12*, 157-173. https://doi.org/10.1162/tacl_a_00638 ↩

.4. Kato. (2026). *How LLMs actually work*. https://www.0xkato.xyz/how-llms-actually-work/. Consultado el 14 de junio de 2026. ↩

.5. Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. https://arxiv.org/abs/2305.13245 ↩

.6. Zhang, B. y Sennrich, R. (2019). Root Mean Square Layer Normalization. *Advances in Neural Information Processing Systems 32*. https://arxiv.org/abs/1910.07467 ↩

.7. Shazeer, N. (2020). GLU Variants Improve Transformer. https://arxiv.org/abs/2002.05202 ↩

.8. Leviathan, Y., Kalman, M. y Matias, Y. (2023). Fast Inference from Transformers via Speculative Decoding. *Proceedings of the 40th International Conference on Machine Learning*. https://arxiv.org/abs/2211.17192 ↩