PUIGFLOWS · GUÍA / GUIDE

Prompt Caching en LLMs: Reduce el Coste y la Latencia de tu Aplicación de IA

Edición ampliada · Julio 2026 · ~36 min de lectura · Español + English

ESPAÑOL

Cada vez que llamas a un LLM, pagas por procesar el prompt completo: instrucciones de sistema, definiciones de herramientas, ejemplos few-shot, documentos de contexto y todo el historial de la conversación. En una aplicación real, la mayor parte de ese contenido es idéntico petición tras petición. El prompt caching existe precisamente para eso: el proveedor guarda el trabajo ya hecho sobre la parte estable del prompt y te cobra una fracción por reutilizarlo. Bien aplicado, reduce el coste de entrada hasta un 90% y la latencia del primer token entre un 30% y un 80%. Mal aplicado, puede incluso salir más caro. Esta guía cubre cómo funciona, cómo estructurar tus prompts para maximizar los aciertos y los errores que anulan la caché sin que te des cuenta.

Qué es el prompt caching y por qué importa

Antes de generar el primer token de salida, el modelo tiene que hacer una pasada completa (prefill) sobre todos los tokens de entrada. Ese prefill produce, para cada capa y cada posición, los tensores clave-valor (la KV cache) que la atención necesita durante la generación. Es la parte más cara y lenta de una petición con prompts largos.

El prompt caching consiste en que el servidor almacene esos tensores para un prefijo del prompt y los reutilice cuando una petición posterior comparte exactamente ese mismo prefijo. No se recomputa nada: el modelo retoma el trabajo donde lo dejó. Por eso los tokens cacheados se facturan mucho más baratos y la respuesta llega antes.

La palabra clave es **prefijo**. La coincidencia es desde el primer token y byte a byte: si cambias un solo carácter al principio del prompt, todo lo que viene después queda invalidado. Esta propiedad dicta toda la estrategia de diseño que veremos a continuación.

Qué ofrece cada proveedor

Los tres grandes proveedores soportan prompt caching, con modelos de control distintos. Los precios citados son de mediados de 2026; verifica siempre la página de precios actual antes de hacer números.

Anthropic (Claude): control explícito

Marcas los puntos de corte con `cache_control` (hasta 4 breakpoints por petición). La escritura en caché con TTL de 5 minutos cuesta 1,25× el precio base de entrada; con TTL de 1 hora, 2×. Las lecturas cuestan en torno al 10% del precio base, y cada lectura refresca el TTL. El punto de equilibrio llega ya en la primera lectura con el TTL de 5 minutos (y en la segunda con el de 1 hora): a partir de ahí, todo es ahorro. Hay un mínimo cacheable de entre 1.024 y 4.096 tokens según el modelo; por debajo de eso, el breakpoint se ignora.

OpenAI: automático

No hay que marcar nada: los prompts de 1.024 tokens o más se cachean automáticamente por prefijo, sin recargo de escritura. El descuento sobre la porción cacheada depende del modelo (entre el 50% y el 90% en la familia GPT-5.x). La caché vive unos minutos tras el último uso. El campo `usage.prompt_tokens_details.cached_tokens` te dice cuánto se sirvió desde caché.

Google (Gemini): implícito y explícito

Gemini 2.5 y posteriores aplican caching implícito automático con descuento sobre los tokens acertados. Además existe el caching explícito (context caching), donde creas un objeto de caché con TTL configurable y pagas el almacenamiento por tiempo, útil para contextos enormes consultados muchas veces.

La regla de oro: estático primero, dinámico al final

Como la coincidencia es por prefijo, el orden de tu prompt determina tu tasa de aciertos. La estructura correcta es siempre la misma, de más estable a más volátil:

1. Definiciones de herramientas (tools)
2. Prompt de sistema con instrucciones y políticas
3. Ejemplos few-shot y documentos de referencia
4. Historial de la conversación
5. El mensaje nuevo del usuario

Un error clásico: empezar el prompt de sistema con la fecha y hora actuales o con el nombre del usuario. Ese dato cambia en cada petición y rompe el prefijo desde el primer token, así que la tasa de aciertos se queda en cero aunque el 95% del prompt sea idéntico. La fecha, el nombre y cualquier dato por petición van al final, normalmente dentro del mensaje del usuario.

Implementación con la API de Anthropic

El ejemplo siguiente cachea un prompt de sistema largo con un breakpoint. La primera llamada paga la escritura; todas las siguientes dentro del TTL leen del prefijo cacheado:

```
import anthropic

client = anthropic.Anthropic()

SYSTEM_PROMPT = """Eres el asistente de soporte de Acme.
[... instrucciones extensas, politicas y ejemplos: ~2.000 tokens ...]"""

def responder(pregunta: str) -> str:
    respuesta = client.messages.create(
        model="claude-sonnet-4-5",
        max_tokens=1024,
        system=[
            {
                "type": "text",
                "text": SYSTEM_PROMPT,
                # Breakpoint: todo lo anterior a este punto se cachea
                "cache_control": {"type": "ephemeral"},
            }
        ],
        messages=[{"role": "user", "content": pregunta}],
    )
    u = respuesta.usage
    print(f"Escritura en cache: {u.cache_creation_input_tokens} tokens")
    print(f"Lectura de cache: {u.cache_read_input_tokens} tokens")
    print(f"Sin cachear: {u.input_tokens} tokens")
    return respuesta.content[0].text
```

En la primera llamada verás todos los tokens en `cache_creation_input_tokens`; en la segunda, en `cache_read_input_tokens`. Si ambos aparecen a cero, el prompt no llega al mínimo cacheable o el breakpoint está mal colocado.

Cacheando la conversación en un agente

En un agente o chatbot, el historial crece turno a turno y cada turno reenvía todo lo anterior. La técnica es marcar el breakpoint en el último bloque del último mensaje: así cacheas todo el historial hasta ese punto, y en el turno siguiente ese prefijo entero es un acierto. El breakpoint avanza con la conversación y solo se paga por los tramos nuevos.

```
def turno_de_agente(historial: list, mensaje: str):
    mensajes = historial + [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": mensaje,
                    # El breakpoint avanza con la conversacion:
                    # cachea todo el historial hasta este punto
                    "cache_control": {"type": "ephemeral"},
                }
            ],
        }
    ]
    return client.messages.create(
        model="claude-sonnet-4-5",
        max_tokens=2048,
        system=SYSTEM_BLOCKS,  # con su propio breakpoint
        tools=TOOLS,           # las tools se cachean como parte del prefijo
        messages=mensajes,
    )
```

Este patrón es el que más dinero ahorra en la práctica: en un bucle agéntico de 20 turnos con herramientas, sin caché pagas el contexto completo 20 veces; con caché pagas una escritura y 19 lecturas al 10%.

OpenAI: caching automático

Con OpenAI no hay breakpoints: basta con respetar el orden estático-primero y comprobar cuánto se sirvió desde caché:

```
from openai import OpenAI

client = OpenAI()

respuesta = client.chat.completions.create(
    model="gpt-5.1",
    messages=[
        # Estable entre peticiones: va primero
        {"role": "system", "content": SYSTEM_PROMPT},
        # Variable: siempre al final
        {"role": "user", "content": pregunta},
    ],
)

detalles = respuesta.usage.prompt_tokens_details
print(f"Tokens servidos desde cache: {detalles.cached_tokens}")
```

Mide tu hit rate o no sabrás si funciona

La caché falla en silencio: si el prefijo no coincide, la API no devuelve un error, simplemente te cobra el precio completo. Por eso conviene registrar las métricas de caché en cada llamada y graficarlas:

```
def metricas_de_cache(usage) -> dict:
    leidos = getattr(usage, "cache_read_input_tokens", 0) or 0
    escritos = getattr(usage, "cache_creation_input_tokens", 0) or 0
    directos = usage.input_tokens
    total = leidos + escritos + directos
    return {
        "hit_rate": leidos / total if total else 0.0,
        "tokens_leidos": leidos,
        "tokens_escritos": escritos,
        "tokens_totales": total,
    }
```

Como referencia real: el equipo de ProjectDiscovery publicó en 2026 que subir su tasa de aciertos del 7% al 84% redujo su factura total de LLM entre un 59% y un 70%. La diferencia no fue cambiar de modelo, sino reordenar prompts y colocar bien los breakpoints.

Errores comunes que invalidan la caché en silencio

- **Contenido dinámico al principio.** Timestamps, IDs de petición o el nombre del usuario en las primeras líneas del sistema rompen el prefijo en cada llamada.
- **Serialización no determinista.** Si generas las definiciones de tools o bloques JSON con orden de claves variable, dos peticiones "iguales" no son idénticas byte a byte y nunca hay acierto.
- **Prompts por debajo del mínimo.** Con menos de 1.024 tokens (más en algunos modelos) no se cachea nada, aunque marques breakpoints.
- **Tráfico demasiado bajo.** Si entre peticiones pasa más tiempo que el TTL, pagas la escritura a 1,25× una y otra vez sin leer nunca: cachear sale más caro que no hacerlo. Sube el TTL a 1 hora o desactívalo.
- **Cambiar de modelo o de tools a mitad de sesión.** La caché es por modelo y por prefijo exacto; cualquier cambio en las definiciones de herramientas invalida todo lo posterior.
- **Recortar o compactar el historial.** Truncar mensajes antiguos para ahorrar contexto reescribe el prefijo y tira la caché entera. Si compactas, hazlo en cortes poco frecuentes y asume el coste de esa petición.
- **No mirar los campos de usage.** Sin métricas, una regresión de caché (un deploy que añade un campo dinámico al sistema) pasa desapercibida durante semanas.

Checklist de buenas prácticas

- Ordena el prompt de más estable a más volátil: tools, sistema, ejemplos, historial, mensaje nuevo.
- Inyecta fecha, usuario y datos por petición al final, nunca al principio del prompt de sistema.
- En agentes, marca el breakpoint en el último mensaje para cachear el historial completo.
- Registra hit rate, tokens leídos y escritos en tus métricas desde el primer día.
- Verifica el punto de equilibrio: con TTL de 5 minutos, la caché es rentable ya desde la primera lectura (con TTL de 1 hora, desde la segunda).
- Usa TTL de 1 hora para prompts caros con tráfico espaciado.
- Congela la serialización de tools y few-shots (mismo orden de claves, mismos espacios) para garantizar igualdad byte a byte.

- Añade un test de humo en CI que haga dos llamadas idénticas y falle si la segunda no lee de caché.

Conclusión

El prompt caching es probablemente la optimización con mejor relación esfuerzo-beneficio disponible hoy en aplicaciones con LLMs: no cambia la calidad de las respuestas, no requiere reentrenar nada y puede recortar la factura de entrada en un orden de magnitud. La clave está en tratar el prompt como una estructura con contrato: prefijo estable, sufijo dinámico, breakpoints bien colocados y métricas que avisen cuando algo lo rompa. Si tu aplicación reenvía el mismo contexto más de una vez por minuto y todavía no cachea, es dinero que estás dejando sobre la mesa.

Edición ampliada: prompt caching a fondo

Las secciones anteriores cubren lo esencial para empezar a cachear hoy mismo. Lo que sigue es el material que marca la diferencia entre "activé la caché" y "diseñé mi aplicación alrededor de la caché": cómo funciona el mecanismo por dentro, las novedades de cada proveedor a mediados de 2026, las matemáticas exactas del ahorro, y los escenarios donde el caching se complica (multi-tenant, RAG, agentes, self-hosted). Todo con código que puedes copiar y adaptar.

Cómo funciona por dentro: la KV cache

Para colocar bien los breakpoints conviene entender qué está guardando exactamente el proveedor. Un transformer genera texto en dos fases. En la fase de **prefill** procesa todos los tokens de entrada en paralelo; en la fase de **decode** genera los tokens de salida uno a uno. Durante el prefill, cada capa de atención calcula, para cada token, dos tensores: la clave (K) y el valor (V). Ese conjunto de tensores es la **KV cache**, y es lo que la atención consulta en cada paso de generación para "recordar" el contexto.

El tamaño de esa estructura es considerable. Por cada token, la KV cache ocupa aproximadamente `2 × capas × cabezas_kv × dimensión_cabeza × bytes_por_valor`. En un modelo denso de 70B parámetros con 80 capas, eso ronda los 300 KB por token en FP16: un prompt de 50.000 tokens ocupa unos 15 GB de memoria de GPU solo en KV cache. Por eso los proveedores no pueden guardar cachés ilimitadas para siempre, por eso existen TTLs, y por eso la escritura en caché tiene recargo: estás alquilando memoria de GPU (o almacenamiento adyacente) durante la vida de la entrada.

Guardar la KV cache de un prefijo y reutilizarla es matemáticamente exacto, no una aproximación: la atención es causal, así que los tensores de los primeros N tokens no dependen de nada posterior. De ahí la regla de coincidencia por prefijo. Si el token 1 cambia, sus K y V cambian, y todos los tokens posteriores atienden sobre valores distintos: no hay nada reutilizable. Los motores de inferencia modernos gestionan esto por bloques (vLLM, por ejemplo, trocea la KV cache en bloques de 16 tokens y los identifica por el hash del prefijo completo hasta ese bloque), lo que permite compartir memoria entre peticiones que comparten prefijo y desalojar por LRU los bloques fríos.

Dos consecuencias prácticas de este diseño:

- **La granularidad no es de un token.** Los aciertos se producen en incrementos de bloque (OpenAI documenta incrementos de 128 tokens a partir del mínimo de 1.024). Un prefijo compartido de 1.150 tokens puede acertar solo 1.024.
- **La caché es por modelo y por configuración.** Los tensores dependen de los pesos: cambiar de modelo, de versión o de parámetros que alteran el preprocesado del prompt (como la configuración de thinking o las definiciones de herramientas) implica una caché nueva.

Anthropic en detalle: caché automática, TTL extendido y desglose de usage

Desde finales de 2025 la API de Anthropic ofrece dos modos de caching, y la recomendación oficial ha cambiado: para la mayoría de los casos ya no hace falta colocar breakpoints a mano.

Caché automática: un solo campo

Con la **caché automática** añades un único `cache_control` a nivel raíz de la petición y el sistema coloca y desplaza los breakpoints por ti a medida que la conversación crece. Es el equivalente gestionado del patrón "breakpoint en el último mensaje" que vimos antes, sin tener que reimplementarlo:

```
respuesta = client.messages.create(
    model="claude-sonnet-4-5",
    max_tokens=2048,
    # Un solo campo: el servidor gestiona los breakpoints
    cache_control={"type": "ephemeral"},
    system=SYSTEM_PROMPT,
    tools=TOOLS,
    messages=historial + [{"role": "user", "content": mensaje}],
)
```

La caché automática es el punto de partida recomendado. Los breakpoints explícitos siguen disponibles (hasta 4 por petición) para control fino: por ejemplo, cachear el prompt de sistema con TTL de 1 hora y el historial con TTL de 5 minutos, o fijar un corte exactamente antes de un bloque que sabes que va a variar.

TTL de 1 hora y desglose del usage

El TTL se elige por breakpoint con el campo `ttl`. La escritura a 1 hora cuesta 2× el precio base (frente a 1,25× de la de 5 minutos), y cada lectura refresca el TTL sin coste de escritura adicional:

```
system=[
    {
        "type": "text",
        "text": SYSTEM_PROMPT,
        "cache_control": {"type": "ephemeral", "ttl": "1h"},
    }
]

# El usage desglosa las escrituras por TTL:
# usage.cache_creation.ephemeral_5m_input_tokens
# usage.cache_creation.ephemeral_1h_input_tokens
# usage.cache_read_input_tokens
```

Ese desglose importa para tus métricas de coste: una escritura de 5 minutos y una de 1 hora no cuestan lo mismo, y si tu panel solo suma `cache_creation_input_tokens` estás mezclando precios distintos.

El punto de equilibrio, con números

Merece la pena corregir una intuición extendida (incluida la versión anterior de esta guía): con el TTL de 5 minutos, la caché es rentable **desde la primera lectura**, no desde la segunda. La cuenta con multiplicadores sobre el precio base: dos peticiones sin caché cuestan 1,0 + 1,0 = 2,0; con caché cuestan 1,25 (escritura) + 0,1 (lectura) = 1,35. Ya has ahorrado un 32%. Con el TTL de 1 hora (2× de escritura), necesitas dos lecturas: 2,0 + 0,1 + 0,1 = 2,2 frente a 3,0 sin caché.

Con los precios vigentes de mediados de 2026, por millón de tokens: Claude Sonnet 4.5 cuesta $3 de entrada base, $3,75 la escritura de 5 minutos, $6 la de 1 hora y $0,30 la lectura. En Haiku 4.5 ($1 base) la lectura queda en $0,10 por millón. Verifica siempre la tabla oficial: Sonnet 5, por ejemplo, tiene precio introductorio ($2/$10) hasta el 31 de agosto de 2026.

Detalles que conviene saber

- **Los multiplicadores se acumulan con otros descuentos.** La Batch API (50% de descuento) se combina con el caching: una lectura de caché dentro de un batch cuesta 0,1 × 0,5 = 5% del precio base de entrada. Para pipelines masivos no urgentes es la combinación más barata posible.
- **La jerarquía del prefijo es tools → system → messages.** Un breakpoint cachea todo lo anterior a él en ese orden. Cambiar una definición de herramienta invalida también system y messages; cambiar solo el último mensaje no invalida nada anterior.
- **El mínimo cacheable depende del modelo** (1.024 tokens en los modelos grandes, 4.096 en algunos ligeros). Los breakpoints por debajo del mínimo se ignoran en silencio.
- **Cambios de tokenizador entre versiones.** Los modelos Claude 4.7 y posteriores usan un tokenizador nuevo que produce en torno a un 30% más de tokens para el mismo texto. Al migrar de modelo, tu mismo prompt cambia de longitud en tokens: recalcula costes y umbrales mínimos, y asume que la caché anterior no aplica.
- **Contexto largo a precio estándar.** Desde Claude 4.6 la ventana de 1M de tokens se factura al precio normal por token, y el caching aplica igual en toda la ventana. Esto abre la puerta al patrón "corpus entero en el prompt + caché" como alternativa a RAG para corpus medianos, que veremos más abajo.

OpenAI en detalle: prompt_cache_key y retención extendida

El caching de OpenAI sigue siendo automático (prefijos de 1.024 tokens o más, aciertos en incrementos de 128), pero desde 2025-2026 hay dos parámetros que conviene conocer porque cambian el resultado en producción.

prompt_cache_key: ayuda al enrutado

La infraestructura de OpenAI enruta cada petición a una máquina según un hash del inicio del prompt (y del proyecto). Si tienes muchos prompts que comparten los primeros cientos de tokens pero divergen después, o mucho tráfico concurrente con el mismo prefijo, el parámetro `prompt_cache_key` afina ese enrutado para que las peticiones que deberían compartir caché aterricen en la misma máquina:

```
respuesta = client.responses.create(
    model="gpt-5.1",
    instructions=SYSTEM_PROMPT,          # estable: va primero
    input=pregunta,                      # variable: al final
    # Agrupa el trafico por agente/tenant para mejorar el enrutado
    prompt_cache_key="soporte-acme-v3",
)
print(respuesta.usage.input_tokens_details.cached_tokens)
```

Dos reglas prácticas: usa una clave por *combinación de prefijo* (por agente, por tenant, por versión de prompt), no por usuario final —una clave por usuario fragmenta el tráfico y no aporta nada si cada usuario tiene poco volumen—, y no superes en mucho las ~15 peticiones por minuto por clave, porque el exceso se desborda a otras máquinas y los aciertos caen.

prompt_cache_retention: de minutos a 24 horas

El parámetro `prompt_cache_retention` controla cuánto vive la caché. El modo clásico `in_memory` mantiene los prefijos entre 5 y 10 minutos de inactividad (máximo una hora). El modo `24h` descarga los tensores KV cifrados a almacenamiento local de la GPU cuando la memoria se llena, y mantiene el prefijo activo hasta 24 horas sin recargo de escritura, lo que cambia por completo la economía del tráfico espaciado: un asistente interno que recibe una consulta cada 20 minutos pasa de no acertar nunca a acertar casi siempre.

```
respuesta = client.responses.create(
    model="gpt-5.1",
    instructions=SYSTEM_PROMPT,
    input=pregunta,
    prompt_cache_key="soporte-acme-v3",
    prompt_cache_retention="24h",
)
```

Desde mayo de 2026, en la familia GPT-5 la retención extendida es además el valor por defecto para organizaciones sin Zero Data Retention; con ZDR activo se mantiene `in_memory`. Si tu organización tiene requisitos estrictos de residencia de datos, revisa qué implica el volcado cifrado a almacenamiento local antes de activarlo. El descuento por lectura varía por modelo (entre el 50% y el 90% en la familia GPT-5.x; en los últimos buques insignia la lectura cuesta el 10% del precio base, igual que en Anthropic).

Gemini en detalle: caché implícita y explícita

Google ofrece dos mecanismos complementarios, y conviene no confundirlos.

Caché implícita: activada por defecto

En todos los modelos Gemini 2.5 y posteriores la caché implícita está activada sin configuración alguna: si tu petición comparte prefijo con otra reciente, el ahorro (90% de descuento sobre los tokens acertados en los modelos 2.5+) se aplica solo. Los mínimos por modelo importan: 2.048 tokens en Gemini 2.5 Flash y 2.5 Pro, y 4.096 en Gemini 3.5 Flash y 3.1 Pro. El campo `usage.total_cached_tokens` (o `cached_content_token_count` según la API) te dice cuántos tokens acertaron. Las recomendaciones oficiales son las mismas que ya conoces: contenido grande y común al principio, y peticiones con el mismo prefijo agrupadas en el tiempo.

Caché explícita: tú pagas el alquiler

La caché explícita (context caching clásico de la API generateContent) invierte el modelo: creas un objeto de caché con el contenido y un TTL que eliges, pagas la escritura una vez y un coste de **almacenamiento por token-hora**, y cada petición que lo referencia paga solo los tokens nuevos:

```
from google import genai
from google.genai import types

client = genai.Client()

cache = client.caches.create(
    model="gemini-2.5-pro",
    config=types.CreateCachedContentConfig(
        display_name="manual-del-producto",
        system_instruction=INSTRUCCIONES,
        contents=[documento_largo],   # p. ej. un PDF de 200k tokens
        ttl="3600s",
    ),
)

respuesta = client.models.generate_content(
    model="gemini-2.5-pro",
    contents="Resume el capitulo 4",
    config=types.GenerateContentConfig(cached_content=cache.name),
)
```

La decisión entre implícita y explícita es puramente económica. La implícita no cuesta nada y no garantiza nada; la explícita garantiza el acierto pero cobra almacenamiento mientras el objeto exista. La explícita compensa cuando un contexto enorme (decenas o cientos de miles de tokens) va a recibir muchas consultas en una ventana de tiempo conocida: un vídeo largo que analizas con veinte preguntas, un contrato que procesas entero, una base de código sobre la que iteras durante una hora. Para tráfico difuso, deja trabajar a la implícita.

Las matemáticas del ahorro: cuándo compensa cachear

Antes de activar TTLs largos o cachés explícitas, haz la cuenta. El modelo de coste cabe en una función:

```
def coste_mensual(
    tokens_prefijo: int,        # parte estable del prompt
    tokens_variables: int,      # parte nueva por peticion
    peticiones_dia: int,
    hit_rate: float,            # fraccion de peticiones que aciertan
    precio_base: float,         # $/MTok de entrada
    mult_escritura: float = 1.25,
    mult_lectura: float = 0.10,
) -> float:
    dias = 30
    aciertos = peticiones_dia * hit_rate
    fallos = peticiones_dia * (1 - hit_rate)
    coste_dia = (
        fallos * (tokens_prefijo * mult_escritura + tokens_variables)
        + aciertos * (tokens_prefijo * mult_lectura + tokens_variables)
    ) * precio_base / 1_000_000
    return coste_dia * dias
```

Ejemplo con números redondos: un asistente de soporte sobre Claude Sonnet 4.5 ($3/MTok de entrada) con 3.500 tokens de prefijo estable (sistema + herramientas + ejemplos), 500 tokens variables por petición y 10.000 peticiones diarias. Sin caché: unos $3.600 al mes solo en entrada. Con un hit rate del 90% y TTL de 5 minutos: unos $700. Con el mismo tráfico en Haiku 4.5 la cuenta baja proporcionalmente, y si además el flujo tolera proceso asíncrono, la Batch API deja la lectura de caché en el 5% del precio base.

Tres conclusiones que salen siempre de esta cuenta:

- **El hit rate manda.** Pasar del 50% al 90% de aciertos ahorra más que cualquier otro cambio, incluida la elección de TTL. Por eso las secciones de este recurso insisten tanto en el orden del prompt y la serialización determinista.
- **El TTL correcto depende del espaciado del tráfico, no del tamaño del prompt.** Peticiones cada <5 minutos: TTL corto. Cada 5-60 minutos: TTL de 1 hora (Anthropic) o retención 24h (OpenAI). Más espaciadas: solo la retención de 24 horas te salva, o directamente no caches.
- **El prefijo pequeño no compensa.** Con 1.200 tokens de prefijo y 2.000 variables, el ahorro máximo teórico es un tercio de la factura. Con 30.000 de prefijo y 500 variables, es casi toda. Prioriza cachear donde el prefijo domina.

Multi-tenant, fan-out y enrutamiento consciente de caché

El hit rate que mides en desarrollo con una sola instancia rara vez sobrevive al despliegue real. Tres fenómenos lo erosionan:

- **Prefijos por tenant.** Si cada cliente tiene su propio prompt de sistema (instrucciones personalizadas, su catálogo, su tono), cada tenant es una caché independiente. Diseña el prompt en capas: primero la parte común a todos los tenants (la más larga), después el bloque por tenant, y al final lo variable. Así la parte común se comparte y solo la capa por tenant se escribe por cliente.
- **Fan-out concurrente.** Si lanzas veinte peticiones en paralelo con el mismo prefijo frío, ninguna ve la caché de las demás y pagas veinte escrituras. El patrón correcto es **calentar la caché**: una petición corta inicial (incluso con `max_tokens=1`) que escriba el prefijo, y después el fan-out, que ya lee. En pipelines batch esto ahorra cifras serias.

- **Balanceadores que reparten "bien".** Un load balancer round-robin sobre múltiples claves de API o múltiples réplicas de un gateway reparte peticiones con el mismo prefijo entre destinos distintos. En OpenAI, `prompt_cache_key` mitiga esto del lado del proveedor; en tu propio gateway, usa hashing consistente sobre un identificador del prefijo (por ejemplo, el hash del prompt de sistema) para que el mismo agente caiga siempre en la misma ruta.

RAG y prompt caching: amigos con condiciones

RAG parece el enemigo natural del caching: cada consulta recupera chunks distintos, y ese contenido variable rompe el prefijo. Pero con tres decisiones de diseño conviven bien:

- **Separa lo estable de lo recuperado.** Instrucciones, formato de respuesta, ejemplos few-shot y definiciones de herramientas van antes del primer chunk recuperado, con su breakpoint. Así el prefijo instruccional (a menudo miles de tokens) siempre acierta aunque los documentos cambien.
- **Ordena los chunks de forma determinista.** Si dos consultas recuperan los mismos documentos, serialízalos siempre igual (ordenados por ID, no por score, que fluctúa). Dos peticiones con los mismos chunks en el mismo orden comparten también esa parte del prefijo. Con orden por score, jamás.
- **En RAG conversacional, recupera una vez por sesión cuando puedas.** Si el usuario va a hacer cinco preguntas sobre el mismo documento, inyectar el documento una vez al principio de la conversación (y cachearlo) es más barato y con mejor latencia que recuperar y reinyectar chunks en cada turno, que invalida el historial cacheado.

Y una alternativa que el contexto largo ha hecho viable: para corpus medianos (hasta unos cientos de miles de tokens) y tráfico concentrado, meter *todo el corpus* en el prompt y cachearlo (con TTL de 1 hora en Anthropic, retención 24h en OpenAI o caché explícita en Gemini) puede ser más simple, más barato y de mejor calidad que un pipeline RAG completo con su infraestructura de embeddings, índices y reranking. Haz la cuenta con la función de coste de la sección anterior antes de descartarlo.

Agentes: compactación, subagentes y herramientas

Los bucles agénticos son el caso donde el caching más ahorra y donde más fácil es romperlo sin querer.

- **Los resultados de herramientas forman parte del prefijo.** Cada `tool_result` que entra en el historial queda cacheado en el turno siguiente. Un agente de 30 pasos con resultados de búsqueda largos acumula decenas de miles de tokens que, bien cacheados, se pagan una vez a precio de escritura y el resto a 10%.
- **La compactación tiene un precio, prográmala.** Resumir o truncar el historial reescribe el prefijo y tira toda la caché. No compactes "cuando haga falta" en mitad de una racha de turnos: hazlo en cortes planificados (por ejemplo, al cruzar el 70% de la ventana de contexto), asume esa petición a precio completo y vuelve a cachear el historial compactado inmediatamente.
- **Subagentes: comparte el prefijo común.** Si un orquestador lanza subagentes con instrucciones parcialmente comunes, estructura los prompts para que la parte compartida sea un prefijo idéntico byte a byte en todos ellos, con la especialización después. Diez subagentes con 4.000 tokens comunes bien ordenados son una escritura y nueve lecturas.
- **Cuidado con las features que editan el contexto.** Cualquier mecanismo que borre resultados de herramientas antiguos o reescriba mensajes (context editing, poda automática de historial) invalida la caché

desde el punto editado. A veces compensa igualmente (contexto más corto = menos tokens en cada turno futuro), pero decídelo con las métricas delante, no por defecto.

Self-hosted: vLLM, SGLang y prefix caching en tu infraestructura

Si sirves modelos abiertos, el prefix caching no es cosa del proveedor: es tuya. Las dos referencias:

- **vLLM** implementa Automatic Prefix Caching (APC): trocea la KV cache en bloques identificados por el hash del prefijo y reutiliza bloques entre peticiones, con desalojo LRU cuando la memoria aprieta. Se activa al lanzar el servidor con `--enable-prefix-caching` y no requiere cambios en el cliente. Las métricas del servidor exponen la tasa de aciertos de prefijo: vigílala igual que vigilarías la de un proveedor.
- **SGLang** va más lejos con RadixAttention: organiza los prefijos en un árbol de radix compartido, de modo que muchas conversaciones que ramifican desde el mismo sistema comparten automáticamente la parte común sin duplicarla en memoria. Para cargas con mucho prefijo común (agentes, few-shot masivo, evaluaciones) la diferencia de throughput es notable.

La lección de infraestructura es la misma que con los proveedores: el enrutado decide el hit rate. Con varias réplicas, un balanceador que ignora el contenido reparte prefijos idénticos entre GPUs distintas y cada una recalcula lo mismo. Los gateways de inferencia modernos ofrecen enrutado consciente de prefijos (prefix-aware routing): enrutan por hash del inicio del prompt para que el mismo prefijo caiga siempre en la misma réplica. Si operas a escala, esa pieza vale más que cualquier ajuste fino del motor.

Seguridad y privacidad

- **Aislamiento por organización.** En los tres grandes proveedores la caché está aislada a nivel de organización: nadie fuera de tu cuenta puede acertar contra tus prefijos. Además, la coincidencia exige el prefijo exacto completo: no existe "acierto parcial" que revele contenido a quien no lo tiene ya.
- **La retención extendida almacena tensores, no texto plano.** El modo de 24 horas de OpenAI descarga tensores KV cifrados a almacenamiento local de GPU y los elimina al expirar; con Zero Data Retention activo, la retención extendida no se usa. Si tienes compromisos contractuales de no persistencia, documenta qué modo usas.
- **No pongas secretos por usuario en prefijos compartidos.** Un token de API o un dato personal dentro del bloque cacheado compartido entre peticiones es un olor de diseño: además de fragmentar la caché, mezcla ámbitos de datos. Lo por-usuario va al final, fuera del prefijo común.
- **El caching no cambia la calidad ni el contenido de las respuestas.** Reutilizar la KV cache produce exactamente el mismo cálculo que reprocesar el prefijo. No hay trade-off de calidad: solo de coste, latencia y arquitectura.

Observabilidad: dashboards y alertas

La regresión de caché típica no es un incidente: es un deploy inocente que añade un campo dinámico al principio del sistema y multiplica la factura por cinco sin que nada "falle". Se detecta con métricas, no con logs de error. Instrumenta cada llamada:

```
from prometheus_client import Counter

TOKENS = Counter(
    "llm_input_tokens_total",
    "Tokens de entrada por tipo de cache",
    ["modelo", "tipo"],  # tipo: leidos | escritos | directos
)

def registrar(modelo: str, usage) -> None:
    TOKENS.labels(modelo, "leidos").inc(
        getattr(usage, "cache_read_input_tokens", 0) or 0)
    TOKENS.labels(modelo, "escritos").inc(
        getattr(usage, "cache_creation_input_tokens", 0) or 0)
    TOKENS.labels(modelo, "directos").inc(usage.input_tokens)
```

Con ese contador, el hit rate es una consulta PromQL y la alerta que de verdad protege tu factura es una comparación con la semana anterior:

```
# Hit rate por modelo (ventana de 15 minutos)
sum by (modelo) (rate(llm_input_tokens_total{tipo="leidos"}[15m]))
/
sum by (modelo) (rate(llm_input_tokens_total[15m]))

# Alerta: el hit rate cayo mas de 20 puntos tras un deploy
(consulta_anterior) < (consulta_anterior offset 1w) - 0.20
```

Añade el hit rate al dashboard que mira el equipo tras cada despliegue, junto a errores y latencia. Una caída brusca tras un deploy casi siempre es un cambio de prompt que rompió el prefijo.

Test de humo en CI

El complemento barato de las métricas: un test que falla el pipeline si alguien rompe la cacheabilidad del prompt principal. Dos llamadas idénticas; la segunda debe leer de caché:

```
import pytest

def test_prompt_principal_es_cacheable(client_real):
    peticion = construir_peticion_de_produccion("ping")

    primera = client_real.messages.create(**peticion)
    segunda = client_real.messages.create(**peticion)

    leidos = segunda.usage.cache_read_input_tokens
    assert leidos > 0, (
        "La segunda llamada no leyo de cache: "
        "¿campo dinamico al principio del prompt? "
        "¿prefijo por debajo del minimo cacheable?"
    )
```

Ejecútalo en CI con el prompt real de producción (construido por el mismo código que sirve tráfico, no una copia). Es el equivalente para costes de un test de humo de despliegue: barato, binario y se dispara justo cuando alguien introduce la regresión, no semanas después en la factura.

Caso práctico: del 12% al 89% de aciertos

Para aterrizar todo lo anterior, la anatomía de una optimización típica. Un asistente interno de soporte técnico: 4.200 tokens de sistema + herramientas, historial medio de 6 turnos, 18.000 peticiones diarias sobre Sonnet 4.5, hit rate inicial del 12%.

- **Diagnóstico.** El desglose de usage mostró escrituras constantes y lecturas casi nulas. Revisando el prompt: la primera línea del sistema incluía `Fecha actual: {timestamp}` con segundos, y las tools se serializaban desde un dict sin orden fijo.
- **Arreglos, por impacto.** (1) El timestamp bajó al mensaje del usuario y se truncó a fecha (sin hora): el prefijo dejó de cambiar por petición. (2) Serialización de tools congelada con claves ordenadas. (3) Breakpoint movido al último mensaje para cachear el historial completo. (4) TTL de 1 hora para el bloque sistema+tools, porque parte del tráfico llegaba espaciado.
- **Resultado.** Hit rate al 89%, coste de entrada un 71% más bajo, y el tiempo al primer token medio bajó de 3,1s a 1,4s en prompts largos, porque el prefill cacheado también recorta latencia. Ningún cambio de modelo, ninguna pérdida de calidad: solo orden y disciplina de prefijo.

Checklist ampliada

- Empieza con la caché automática (Anthropic) o el caching por defecto (OpenAI, Gemini); pasa a breakpoints explícitos solo cuando tengas un motivo concreto.
- Registra el desglose completo de usage (lecturas, escrituras 5m, escrituras 1h, directos) y grafica el hit rate por modelo y por agente.
- Alerta sobre caídas del hit rate comparadas con la semana anterior, no sobre umbrales absolutos.
- En multi-tenant, estructura el prompt en capas (común → tenant → variable) y usa hashing consistente o `prompt_cache_key` para el enrutado.
- Calienta la caché antes de un fan-out concurrente con una petición corta.
- En RAG, cachea las instrucciones, ordena los chunks de forma determinista y considera corpus-en-prompt para corpus medianos con tráfico concentrado.
- Programa la compactación del historial en cortes planificados y re-cachea inmediatamente.
- Elige TTL por el espaciado del tráfico: <5 min → 5m; 5-60 min → 1h o retención 24h; más → 24h o no cachear.
- Haz la cuenta con la función de coste antes de activar cachés explícitas o TTLs caros.
- Un test de humo en CI que verifique que la segunda llamada idéntica lee de caché.

Preguntas frecuentes

¿El caching afecta a la calidad de las respuestas? No. Reutilizar la KV cache es matemáticamente idéntico a reprocesar el prefijo. La respuesta puede variar entre llamadas por el muestreo, pero no por la caché.

¿Puedo cachear imágenes y PDFs? Sí: los bloques de imagen y documento forman parte del prefijo y se cachean igual que el texto. Ten en cuenta que cambiar un solo píxel o re-codificar el archivo produce bytes distintos y rompe el acierto.

¿Qué pasa si dos peticiones concurrentes escriben el mismo prefijo? Ambas pagan escritura; una de las entradas prevalece. Es inofensivo pero caro a escala: usa el patrón de calentamiento antes del fan-out.

¿El streaming cambia algo? No para el caching: los aciertos se determinan en el prefill, antes del primer token emitido. De hecho, el beneficio de latencia del caching se nota sobre todo en el tiempo al primer token de respuestas en streaming.

¿Cachear conversaciones de usuarios distintos juntos? Solo comparten caché las partes byte-idénticas: el sistema y las tools comunes sí; el historial de cada usuario, no. Eso es exactamente lo que quieres: lo común se comparte, lo privado no.

¿Por qué mi hit rate es bueno en local y malo en producción? Casi siempre: enrutado (varias réplicas o claves repartiendo el mismo prefijo), concurrencia (escrituras duplicadas en frío) o tráfico real más espaciado que el TTL. Las tres tienen sección propia arriba.

¿Los "prompts del sistema" de los frameworks rompen la caché? Pueden. Frameworks que inyectan metadatos variables (IDs de ejecución, fechas) al principio del prompt ensamblado son una fuente clásica de regresiones. Inspecciona el prompt final que sale por la red, no el que crees que envías.

¿Merece la pena en prompts pequeños? Por debajo del mínimo cacheable (1.024-4.096 tokens según modelo y proveedor) no se cachea nada. Justo por encima, el ahorro es proporcional al peso del prefijo: con prefijos de menos de ~2.000 tokens rara vez notarás la diferencia en la factura, aunque la latencia sí mejora algo.

¿Cómo interactúa el caching con salida estructurada y JSON mode? Los esquemas de salida estructurada (response_format, output schemas) forman parte de la configuración de la petición: cambiarlos invalida el prefijo igual que cambiar las tools. Mantén una versión estable del esquema por endpoint y versiónala explícitamente; si necesitas variantes por caso, trátalas como agentes distintos con su propia caché y su propia clave de enrutado.

Conclusión de la edición ampliada

El prompt caching empezó como un truco de facturación y en 2026 es una decisión de arquitectura: condiciona cómo ordenas los prompts, cómo enrutas el tráfico, cómo diseñas agentes y RAG, e incluso si necesitas RAG. La buena noticia es que casi todo el valor se captura con disciplina barata: prefijo estable, serialización congelada, TTL acorde al tráfico, métricas desde el primer día y un test de humo que vigile la puerta. Lo demás —enrutado consciente de caché, cachés explícitas, corpus-en-prompt— son optimizaciones que ya sabrás justificar con números cuando el volumen las pida.

Depuración sistemática: cuando el hit rate no cuadra

Tarde o temprano mirarás el dashboard y el hit rate no será el que esperabas. En lugar de tocar cosas al azar, sigue un orden fijo. Cada paso descarta una familia entera de causas:

- **1. Captura el prompt real que sale por la red.** No el template, ni el que crees que construyes: el payload final. Un proxy local, un interceptor en el SDK o un simple log del JSON serializado. La mayoría de los misterios se resuelven aquí: un middleware que añade un header de contexto, un framework que inyecta la fecha, una librería que reordena claves.

- **2. Compara dos payloads consecutivos byte a byte.** Guarda dos peticiones que deberían compartir prefijo y haz un diff textual. El primer byte diferente marca dónde muere tu prefijo. Todo lo anterior a ese punto es cacheable; todo lo posterior, no. Si el primer diff aparece en la línea 3 del prompt de sistema, ya sabes qué mover al final.
- **3. Verifica el mínimo cacheable.** Cuenta los tokens del prefijo estable (la API de conteo de tokens del proveedor, no una estimación por caracteres). Si está por debajo del mínimo del modelo, no hay nada que depurar: o engordas el prefijo (por ejemplo, moviendo ejemplos few-shot al sistema) o aceptas que ese endpoint no cachea.
- **4. Revisa el desglose de usage, no solo el hit rate.** Escrituras constantes con lecturas nulas: el prefijo cambia en cada petición (vuelve al paso 2). Escrituras y lecturas alternándose: el TTL expira entre peticiones (tráfico más espaciado que el TTL) o el enrutado reparte el tráfico entre cachés distintas. Todo a cero: el breakpoint está mal colocado o por debajo del mínimo.
- **5. Aísla el enrutado.** Reproduce el tráfico desde un solo proceso, en serie, con un solo destino. Si el hit rate se dispara en el experimento y se hunde en producción, el problema es de distribución (réplicas, claves múltiples, fan-out concurrente), no de contenido.

Este orden importa porque los síntomas engañan: un hit rate del 40% puede ser un prompt perfecto con enrutado malo, o un prompt roto con enrutado perfecto, y las soluciones no se parecen en nada.

Comparativa rápida: elegir estrategia por proveedor

Los tres proveedores convergen en la idea (prefijos exactos, lecturas con gran descuento) pero difieren en el control que te dan y en dónde están las trampas:

- **Anthropic** te da el control más fino: breakpoints explícitos con TTL por bloque, caché automática si no quieres pensar, desglose de usage por tipo de escritura y acumulación con Batch API. A cambio, la escritura tiene recargo: cachear tráfico que nunca vuelve a llegar cuesta dinero. Es el modelo mental más claro para agentes complejos con historial largo.
- **OpenAI** lo hace todo automático y sin recargo de escritura, así que activarlo no tiene coste de oportunidad: el peor caso es no ahorrar. El control se ejerce por los laterales, con `prompt_cache_key` para el enrutado y `prompt_cache_retention` para la ventana temporal. La trampa típica es el enrutado con tráfico alto y prefijos parecidos.
- **Gemini** separa los dos casos de uso: la implícita cubre el tráfico general sin que hagas nada, y la explícita es en la práctica un producto distinto —alquiler de contexto por horas— ideal para "muchas preguntas sobre un contexto enorme". La trampa es olvidar borrar cachés explícitas que ya no usas: el almacenamiento se factura mientras existan.

Si construyes multi-proveedor, diseña para el mínimo común: prefijo estable byte a byte y contenido dinámico al final funcionan óptimamente en los tres. Los ajustes específicos (breakpoints, claves de enrutado, cachés explícitas) pueden vivir detrás de tu capa de abstracción de proveedor.

Latencia: el segundo dividendo

El ahorro de coste es el titular, pero en muchas aplicaciones el beneficio decisivo del caching es la latencia. El prefill es proporcional a la longitud del prompt: con 50.000 tokens de contexto, el tiempo al primer token puede

superar varios segundos, y ese coste se paga en cada turno de la conversación. Con el prefijo cacheado, el modelo solo hace prefill de los tokens nuevos.

Consecuencias prácticas:

- **Mide el TTFT por separado para aciertos y fallos.** Un histograma único mezcla dos distribuciones distintas y no te dice nada. Etiqueta la métrica de latencia con `cache_hit=true/false` y compara: la diferencia es tu argumento de negocio para invertir en hit rate.
- **El caching hace viables prompts más ricos.** Sin caché, cada ejemplo few-shot que añades encarece y ralentiza todas las peticiones. Con caché, el coste marginal de un sistema de 8.000 tokens frente a uno de 2.000 es casi nulo a partir de la segunda petición. Muchos equipos recortan calidad de prompt para ahorrar; con buen caching, esa presión desaparece.
- **En UX conversacional, el acierto se nota.** La diferencia entre 3 segundos y 1 segundo al primer token es la diferencia entre "está pensando" y "responde". Si tu producto compite en sensación de velocidad, el hit rate es una métrica de producto, no solo de costes.

Migración: activar caching en una aplicación existente

Activar caching en una app que ya está en producción sin romper nada tiene un orden razonable:

- **Semana 0: mide sin tocar.** Añade el registro del desglose de usage a todas las llamadas. En OpenAI y Gemini el caching ya está activo por defecto: descubre tu hit rate accidental actual. Suele ser revelador (y bajo).
- **Semana 1: audita y reordena.** Con los payloads reales delante, identifica contenido dinámico en posiciones tempranas y muévelo al final. Congela la serialización de tools y ejemplos (orden de claves, espacios). Este paso no requiere tocar la configuración de caché y suele duplicar o triplicar el hit rate por sí solo.
- **Semana 2: activa el control explícito donde aplique.** En Anthropic, añade la caché automática (o breakpoints si tienes un motivo). En OpenAI, añade `prompt_cache_key` por agente y evalúa la retención de 24h para tráfico espaciado. En Gemini, evalúa caché explícita solo para los casos de contexto gigante reutilizado.
- **Semana 3: protege lo ganado.** Test de humo en CI, alerta de caída de hit rate, y una línea en la plantilla de PR: "¿este cambio toca el prompt de sistema o las tools? → revisar impacto en caché". Las regresiones de caché las introducen compañeros bienintencionados en PRs que nadie relacionó con costes.

El error de migración más común es empezar por el paso 2 (tocar configuración) sin el paso 0 (medir): sin línea base no puedes demostrar la mejora, y sin los payloads reales no sabes qué reordenar.

Anti-patrones que vemos repetirse

Para cerrar, la lista negra. Si te reconoces en alguno, es la primera tarea del lunes:

- **El prompt "personalizado" que no personaliza nada.** Interpolar el nombre del usuario en la primera frase del sistema ("Eres el asistente de Marta") por cortesía cosmética, a costa de una caché por usuario. La personalización real va en el mensaje, no en el prefijo.

- **El A/B test de prompts sin aislar.** Alternar dos variantes de sistema sobre el mismo tráfico parte el hit rate por la mitad para las dos. Si haces experimentos de prompt, enruta cada variante de forma pegajosa (el mismo usuario siempre a la misma variante) y mide el coste por variante.
- **Regenerar las tools por petición.** Construir las definiciones de herramientas dinámicamente (filtrando por permisos del usuario, por ejemplo) produce prefijos distintos por usuario. Si los conjuntos de permisos son pocos, genera una variante estable por rol; si son muchos, plantéate si las tools de más pueden viajar igualmente y controlarse en ejecución.
- **Cachear con TTL de 1 hora "por si acaso".** El doble de coste de escritura solo se amortiza si las lecturas llegan pasados los 5 minutos. Con tráfico denso, el TTL corto más el refresco automático en cada lectura mantiene la caché viva igual, a mitad de precio de escritura.
- **Confiar en el dashboard del proveedor como única métrica.** Las consolas de facturación agregan por día y por clave: sirven para confirmar la tendencia, no para detectar la regresión del deploy de esta mañana. La métrica en tu telemetría, con resolución de minutos y etiquetada por versión de despliegue, es la que te avisa a tiempo.

Nada de esta lista es exótico: son decisiones razonables tomadas sin tener la caché en mente. Esa es la moraleja de toda la guía. El prompt caching no pide heroicidades; pide que el prefijo sea un contrato estable y que alguien lo esté midiendo.

Gobernanza de costes: del hit rate al presupuesto

En equipos con varias features sobre LLMs, el caching bien medido habilita algo más valioso que el ahorro puntual: atribución de costes. Si etiquetas cada llamada con la feature que la origina (una cabecera interna, un campo en tu telemetría), puedes construir el coste real por feature con la caché ya descontada, y las conversaciones cambian de tono: en lugar de "la factura de IA subió un 40%", discutes "el autocompletado cuesta $0,0004 por invocación y el resumen de documentos $0,09, ¿cuál optimizamos?".

Tres prácticas concretas que salen casi gratis una vez tienes las métricas de caché:

- **Coste por petición efectiva como SLO económico.** Define para cada feature un coste objetivo por invocación (tokens directos + escrituras + lecturas, a precios reales) y alerta cuando se desvíe. Las regresiones de prompt, de caché o de verbosidad del modelo aparecen todas en esa única métrica.
- **Presupuesto de escrituras.** Las escrituras de caché son inversión: solo se recuperan si llegan lecturas. Un ratio lecturas/escrituras persistentemente bajo (<2) en una feature indica caché mal dimensionada: TTL corto para su tráfico, prefijos fragmentados o tráfico demasiado escaso para cachear.
- **Revisión mensual de TTLs.** El patrón de tráfico cambia con el producto: una feature que pasó de uso continuo a esporádico merece pasar de TTL corto a retención larga (o a no cachear). Diez minutos al mes mirando el desglose por feature evitan pagar escrituras que nunca se leen.

Nada de esto requiere herramientas nuevas: son las mismas métricas de usage de esta guía, agregadas con una etiqueta más. La diferencia está en convertirlas en rutina de equipo en lugar de en curiosidad del dashboard.

Glosario rápido

- **Prefill:** la pasada inicial del modelo sobre todos los tokens de entrada, previa a generar el primer token de salida. Es lo que el caching evita repetir.
- **KV cache:** los tensores clave-valor que el prefill produce por capa y por token; el objeto que el proveedor guarda y reutiliza.
- **Prefijo:** los primeros N tokens de la petición, comparados byte a byte desde el inicio. La unidad de coincidencia del caching.
- **Breakpoint:** marca explícita (Anthropic `cache_control`) que delimita hasta dónde cachear.
- **TTL:** tiempo de vida de la entrada de caché desde su último uso; las lecturas lo refrescan.
- **Hit rate:** fracción de tokens de entrada servidos desde caché sobre el total. La métrica que resume toda tu estrategia.
- **Cache warming:** petición inicial deliberada que escribe el prefijo en caché antes de un fan-out concurrente.
- **Enrutado consciente de prefijos:** dirigir peticiones con el mismo prefijo al mismo servidor o réplica para maximizar la reutilización.

ENGLISH

Prompt Caching in LLMs: Cut the Cost and Latency of Your AI Application

Extended edition · July 2026 · ~36 min read

Every time you call an LLM, you pay to process the entire prompt: system instructions, tool definitions, few-shot examples, context documents, and the full conversation history. In a real application, most of that content is identical from one request to the next. Prompt caching exists precisely for this: the provider stores the work already done on the stable part of the prompt and charges you a fraction to reuse it. Applied well, it cuts input costs by up to 90% and time-to-first-token by 30–80%. Applied badly, it can actually cost you more. This guide covers how it works, how to structure your prompts to maximize hits, and the mistakes that silently defeat the cache.

What prompt caching is and why it matters

Before generating the first output token, the model must run a full pass (prefill) over every input token. That prefill produces, for each layer and each position, the key-value tensors (the KV cache) that attention needs during generation. It is the most expensive and slowest part of a request with long prompts.

Prompt caching means the server stores those tensors for a prefix of the prompt and reuses them when a later request shares exactly that same prefix. Nothing is recomputed: the model picks up where it left off. That is why cached tokens are billed far cheaper and responses arrive sooner.

The key word is **prefix**. Matching starts at the first token and is byte-for-byte: change a single character near the start of the prompt and everything after it is invalidated. This property dictates the entire design strategy that follows.

What each provider offers

All three major providers support prompt caching, with different control models. Prices quoted are as of mid-2026; always check the current pricing page before running the numbers.

Anthropic (Claude): explicit control

You mark cut points with `cache_control` (up to 4 breakpoints per request). A cache write with a 5-minute TTL costs 1.25× the base input price; with a 1-hour TTL, 2×. Reads cost around 10% of the base price, and every read refreshes the TTL. Break-even arrives on the very first read with the 5-minute TTL (and on the second read with the 1-hour TTL): from there on it is all savings. There is a cacheable minimum of 1,024–4,096 tokens depending on the model; below that, the breakpoint is ignored.

OpenAI: automatic

Nothing to mark: prompts of 1,024 tokens or more are cached automatically by prefix, with no write surcharge. The discount on the cached portion depends on the model (between 50% and 90% on the GPT-5.x family). The cache lives for a few minutes after last use. The `usage.prompt_tokens_details.cached_tokens` field tells you how much was served from cache.

Google (Gemini): implicit and explicit

Gemini 2.5 and later apply automatic implicit caching with a discount on matched tokens. There is also explicit caching (context caching), where you create a cache object with a configurable TTL and pay for storage by time — useful for huge contexts queried many times.

The golden rule: static first, dynamic last

Since matching is by prefix, the order of your prompt determines your hit rate. The correct structure is always the same, from most stable to most volatile:

1. Tool definitions
2. System prompt with instructions and policies
3. Few-shot examples and reference documents
4. Conversation history
5. The new user message

A classic mistake: starting the system prompt with the current date and time or the user's name. That value changes on every request and breaks the prefix from the very first token, so your hit rate stays at zero even though 95% of the prompt is identical. The date, the name, and any per-request data go at the end, usually inside the user message.

Implementation with the Anthropic API

The following example caches a long system prompt with one breakpoint. The first call pays the write; every subsequent call within the TTL reads from the cached prefix:

```
import anthropic

client = anthropic.Anthropic()

SYSTEM_PROMPT = """You are Acme's support assistant.
[... long instructions, policies and examples: ~2,000 tokens ...]"""

def answer(question: str) -> str:
    response = client.messages.create(
        model="claude-sonnet-4-5",
        max_tokens=1024,
        system=[
            {
                "type": "text",
                "text": SYSTEM_PROMPT,
                # Breakpoint: everything before this point gets cached
                "cache_control": {"type": "ephemeral"},
            }
        ],
        messages=[{"role": "user", "content": question}],
    )
    u = response.usage
    print(f"Cache write: {u.cache_creation_input_tokens} tokens")
    print(f"Cache read: {u.cache_read_input_tokens} tokens")
    print(f"Uncached: {u.input_tokens} tokens")
    return response.content[0].text
```

On the first call you will see all tokens under `cache_creation_input_tokens`; on the second, under `cache_read_input_tokens`. If both come back zero, the prompt is below the cacheable minimum or the breakpoint is misplaced.

Caching the conversation in an agent

In an agent or chatbot, the history grows turn by turn and each turn resends everything before it. The technique is to mark the breakpoint on the last block of the last message: that caches the entire history up to that point, and on the next turn that whole prefix is a hit. The breakpoint advances with the conversation and you only pay for the new segments.

```
def agent_turn(history: list, message: str):
    messages = history + [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": message,
                    # The breakpoint advances with the conversation:
                    # it caches the entire history up to this point
                    "cache_control": {"type": "ephemeral"},
                }
            ],
        }
    ]
    return client.messages.create(
        model="claude-sonnet-4-5",
        max_tokens=2048,
        system=SYSTEM_BLOCKS,  # with its own breakpoint
        tools=TOOLS,           # tools are cached as part of the prefix
        messages=messages,
    )
```

This pattern saves the most money in practice: in a 20-turn agentic loop with tools, without caching you pay for the full context 20 times; with caching you pay one write and 19 reads at 10%.

OpenAI: automatic caching

With OpenAI there are no breakpoints: just respect the static-first ordering and check how much was served from cache:

```
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
    model="gpt-5.1",
    messages=[
        # Stable across requests: goes first
        {"role": "system", "content": SYSTEM_PROMPT},
        # Variable: always last
        {"role": "user", "content": question},
    ],
)

details = response.usage.prompt_tokens_details
print(f"Tokens served from cache: {details.cached_tokens}")
```

Measure your hit rate or you will never know it works

The cache fails silently: if the prefix does not match, the API returns no error — it simply charges you full price. That is why you should log cache metrics on every call and graph them:

```
def cache_metrics(usage) -> dict:
    reads = getattr(usage, "cache_read_input_tokens", 0) or 0
    writes = getattr(usage, "cache_creation_input_tokens", 0) or 0
    direct = usage.input_tokens
    total = reads + writes + direct
    return {
        "hit_rate": reads / total if total else 0.0,
        "tokens_read": reads,
        "tokens_written": writes,
        "tokens_total": total,
    }
```

As a real-world reference: the ProjectDiscovery team reported in 2026 that raising their hit rate from 7% to 84% cut their total LLM bill by 59–70%. The difference was not switching models — it was reordering prompts and placing breakpoints correctly.

Common mistakes that silently invalidate the cache

- **Dynamic content at the start.** Timestamps, request IDs, or the user's name in the first lines of the system prompt break the prefix on every call.
- **Non-deterministic serialization.** If you generate tool definitions or JSON blocks with variable key order, two "identical" requests are not byte-for-byte equal and you never get a hit.
- **Prompts below the minimum.** Under 1,024 tokens (more on some models) nothing is cached, even if you set breakpoints.
- **Traffic that is too sparse.** If more time passes between requests than the TTL, you pay the 1.25× write over and over without ever reading: caching costs more than not caching. Move to the 1-hour TTL or turn it off.
- **Switching models or tools mid-session.** The cache is per model and per exact prefix; any change to tool definitions invalidates everything after it.
- **Trimming or compacting the history.** Truncating old messages to save context rewrites the prefix and throws away the whole cache. If you compact, do it at infrequent cut points and accept the cost of that one request.
- **Ignoring the usage fields.** Without metrics, a cache regression (a deploy that adds one dynamic field to the system prompt) goes unnoticed for weeks.

Best-practices checklist

- Order the prompt from most stable to most volatile: tools, system, examples, history, new message.
- Inject the date, user, and per-request data at the end — never at the top of the system prompt.
- In agents, set the breakpoint on the last message to cache the full history.
- Log hit rate, tokens read, and tokens written in your metrics from day one.
- Check your break-even: with the 5-minute TTL, caching pays off from the first read (with the 1-hour TTL, from the second).
- Use the 1-hour TTL for expensive prompts with spaced-out traffic.

- Freeze the serialization of tools and few-shots (same key order, same whitespace) to guarantee byte-for-byte equality.
- Add a smoke test in CI that makes two identical calls and fails if the second one does not read from cache.

Conclusion

Prompt caching is arguably the best effort-to-payoff optimization available today for LLM applications: it does not change response quality, requires no retraining, and can cut your input bill by an order of magnitude. The key is to treat the prompt as a structure with a contract: stable prefix, dynamic suffix, well-placed breakpoints, and metrics that alert you when something breaks it. If your application resends the same context more than once a minute and still is not caching, that is money left on the table.

Extended edition: prompt caching in depth

The sections above cover what you need to start caching today. What follows is the material that separates "I turned caching on" from "I designed my application around the cache": how the mechanism works under the hood, what each provider offers as of mid-2026, the exact math of the savings, and the scenarios where caching gets tricky (multi-tenant, RAG, agents, self-hosted). All of it with code you can copy and adapt.

How it works under the hood: the KV cache

To place breakpoints well, it helps to understand exactly what the provider is storing. A transformer generates text in two phases. In the **prefill** phase it processes all input tokens in parallel; in the **decode** phase it generates output tokens one at a time. During prefill, each attention layer computes two tensors for every token: the key (K) and the value (V). That collection of tensors is the **KV cache**, and it is what attention consults at every generation step to "remember" the context.

The structure is large. Per token, the KV cache takes roughly `2 × layers × kv_heads × head_dim × bytes_per_value`. In a dense 70B-parameter model with 80 layers, that is around 300 KB per token in FP16: a 50,000-token prompt occupies about 15 GB of GPU memory in KV cache alone. That is why providers cannot keep unlimited caches forever, why TTLs exist, and why cache writes carry a surcharge: you are renting GPU memory (or adjacent storage) for the lifetime of the entry.

Storing the KV cache of a prefix and reusing it is mathematically exact, not an approximation: attention is causal, so the tensors for the first N tokens depend on nothing that comes after them. Hence the prefix-matching rule. If token 1 changes, its K and V change, and every later token attends over different values: nothing is reusable. Modern inference engines manage this in blocks (vLLM, for example, slices the KV cache into 16-token blocks identified by the hash of the full prefix up to that block), which lets requests that share a prefix share memory, with LRU eviction for cold blocks.

Two practical consequences of this design:

- **Granularity is not one token.** Hits happen in block increments (OpenAI documents 128-token increments beyond the 1,024 minimum). A shared prefix of 1,150 tokens may only score a 1,024-token hit.

- **The cache is per model and per configuration.** The tensors depend on the weights: switching models, versions, or parameters that alter prompt preprocessing (like thinking configuration or tool definitions) means a fresh cache.

Anthropic in detail: automatic caching, extended TTL, and the usage breakdown

Since late 2025 the Anthropic API offers two caching modes, and the official recommendation has changed: for most use cases you no longer need to place breakpoints by hand.

Automatic caching: a single field

With **automatic caching** you add a single `cache_control` at the top level of the request and the system places and advances breakpoints for you as the conversation grows. It is the managed equivalent of the "breakpoint on the last message" pattern we saw earlier, without having to reimplement it:

```
response = client.messages.create(
    model="claude-sonnet-4-5",
    max_tokens=2048,
    # One field: the server manages breakpoints for you
    cache_control={"type": "ephemeral"},
    system=SYSTEM_PROMPT,
    tools=TOOLS,
    messages=history + [{"role": "user", "content": message}],
)
```

Automatic caching is the recommended starting point. Explicit breakpoints remain available (up to 4 per request) for fine-grained control: for example, caching the system prompt with a 1-hour TTL and the history with a 5-minute TTL, or pinning a cut right before a block you know will vary.

The 1-hour TTL and the usage breakdown

The TTL is chosen per breakpoint with the `ttl` field. A 1-hour write costs 2× the base price (versus 1.25× for the 5-minute write), and every read refreshes the TTL at no extra write cost:

```
system=[
    {
        "type": "text",
        "text": SYSTEM_PROMPT,
        "cache_control": {"type": "ephemeral", "ttl": "1h"},
    }
]

# usage breaks writes down by TTL:
# usage.cache_creation.ephemeral_5m_input_tokens
# usage.cache_creation.ephemeral_1h_input_tokens
# usage.cache_read_input_tokens
```

That breakdown matters for your cost metrics: a 5-minute write and a 1-hour write are not billed the same, and if your dashboard only sums `cache_creation_input_tokens` you are mixing different prices.

Break-even, with numbers

A widespread intuition is worth correcting (including in the earlier version of this guide): with the 5-minute TTL, caching pays off **from the first read**, not the second. The math with multipliers over the base price: two uncached requests cost 1.0 + 1.0 = 2.0; with caching they cost 1.25 (write) + 0.1 (read) = 1.35. You have already saved 32%. With the 1-hour TTL (2× write), you need two reads: 2.0 + 0.1 + 0.1 = 2.2 versus 3.0 uncached.

At mid-2026 prices, per million tokens: Claude Sonnet 4.5 costs $3 base input, $3.75 for the 5-minute write, $6 for the 1-hour write, and $0.30 per read. On Haiku 4.5 ($1 base) reads come to $0.10 per million. Always verify the official table: Sonnet 5, for instance, has introductory pricing ($2/$10) through August 31, 2026.

Details worth knowing

- **The multipliers stack with other discounts.** The Batch API (50% off) combines with caching: a cache read inside a batch costs 0.1 × 0.5 = 5% of the base input price. For non-urgent bulk pipelines this is the cheapest combination available.
- **The prefix hierarchy is tools → system → messages.** A breakpoint caches everything before it in that order. Changing a tool definition also invalidates system and messages; changing only the last message invalidates nothing before it.
- **The cacheable minimum depends on the model** (1,024 tokens on the larger models, 4,096 on some lighter ones). Breakpoints below the minimum are silently ignored.
- **Tokenizer changes between versions.** Claude 4.7 and later models use a new tokenizer that produces roughly 30% more tokens for the same text. When you migrate models, the same prompt changes length in tokens: recompute costs and minimum thresholds, and assume the old cache does not apply.
- **Long context at standard pricing.** Since Claude 4.6 the 1M-token context window is billed at the normal per-token rate, and caching applies across the full window. This opens the door to the "entire corpus in the prompt + cache" pattern as an alternative to RAG for medium corpora, which we cover below.

OpenAI in detail: prompt_cache_key and extended retention

OpenAI's caching is still automatic (prefixes of 1,024 tokens or more, hits in 128-token increments), but since 2025-2026 there are two parameters worth knowing because they change outcomes in production.

prompt_cache_key: helping the routing

OpenAI's infrastructure routes each request to a machine based on a hash of the start of the prompt (and the project). If you have many prompts that share their first few hundred tokens but diverge afterwards, or heavy concurrent traffic on the same prefix, the `prompt_cache_key` parameter sharpens that routing so requests that should share a cache land on the same machine:

```
response = client.responses.create(
    model="gpt-5.1",
    instructions=SYSTEM_PROMPT,          # stable: goes first
    input=question,                      # variable: at the end
    # Group traffic by agent/tenant to improve routing
    prompt_cache_key="support-acme-v3",
)
print(response.usage.input_tokens_details.cached_tokens)
```

Two practical rules: use one key per *prefix combination* (per agent, per tenant, per prompt version), not per end user — a key per user fragments traffic and adds nothing if each user has low volume — and do not push far beyond ~15 requests per minute per key, because the excess spills over to other machines and hits drop.

prompt_cache_retention: from minutes to 24 hours

The `prompt_cache_retention` parameter controls how long the cache lives. The classic `in_memory` mode keeps prefixes alive for 5–10 minutes of inactivity (one hour maximum). The `24h` mode offloads encrypted KV tensors to GPU-local storage when memory fills up and keeps the prefix active for up to 24 hours with no write surcharge, which completely changes the economics of spaced-out traffic: an internal assistant that gets one query every 20 minutes goes from never hitting to almost always hitting.

```
response = client.responses.create(
    model="gpt-5.1",
    instructions=SYSTEM_PROMPT,
    input=question,
    prompt_cache_key="support-acme-v3",
    prompt_cache_retention="24h",
)
```

Since May 2026, on the GPT-5 family extended retention is also the default for organizations without Zero Data Retention; with ZDR enabled it stays `in_memory`. If your organization has strict data-residency requirements, review what the encrypted offload to local storage implies before enabling it. The read discount varies by model (between 50% and 90% across the GPT-5.x family; on the latest flagships reads cost 10% of the base price, same as Anthropic).

Gemini in detail: implicit and explicit caching

Google offers two complementary mechanisms, and it pays not to confuse them.

Implicit caching: on by default

On all Gemini 2.5 and later models, implicit caching is enabled with zero configuration: if your request shares a prefix with a recent one, the savings (a 90% discount on matched tokens on 2.5+ models) apply on their own. The per-model minimums matter: 2,048 tokens on Gemini 2.5 Flash and 2.5 Pro, and 4,096 on Gemini 3.5 Flash and 3.1 Pro. The `usage.total_cached_tokens` field (or `cached_content_token_count` depending on the API) tells you how many tokens hit. The official recommendations are the ones you already know: large, common content at the beginning, and same-prefix requests grouped in time.

Explicit caching: you pay the rent

Explicit caching (the classic context caching of the generateContent API) inverts the model: you create a cache object with the content and a TTL of your choosing, pay the write once plus a **storage cost per token-hour**, and every request that references it pays only for the new tokens:

```
from google import genai
from google.genai import types

client = genai.Client()

cache = client.caches.create(
    model="gemini-2.5-pro",
    config=types.CreateCachedContentConfig(
        display_name="product-manual",
        system_instruction=INSTRUCTIONS,
        contents=[long_document],   # e.g. a 200k-token PDF
        ttl="3600s",
    ),
)

response = client.models.generate_content(
    model="gemini-2.5-pro",
    contents="Summarize chapter 4",
    config=types.GenerateContentConfig(cached_content=cache.name),
)
```

The choice between implicit and explicit is purely economic. Implicit costs nothing and guarantees nothing; explicit guarantees the hit but bills storage for as long as the object exists. Explicit pays off when a huge context (tens or hundreds of thousands of tokens) will receive many queries within a known time window: a long video you analyze with twenty questions, a contract you process end to end, a codebase you iterate on for an hour. For diffuse traffic, let implicit do its job.

The math of the savings: when caching pays off

Before enabling long TTLs or explicit caches, run the numbers. The cost model fits in one function:

```
def monthly_cost(
    prefix_tokens: int,         # stable part of the prompt
    variable_tokens: int,       # new content per request
    requests_per_day: int,
    hit_rate: float,            # fraction of requests that hit
    base_price: float,          # $/MTok input
    write_mult: float = 1.25,
    read_mult: float = 0.10,
) -> float:
    days = 30
    hits = requests_per_day * hit_rate
    misses = requests_per_day * (1 - hit_rate)
    daily_cost = (
        misses * (prefix_tokens * write_mult + variable_tokens)
        + hits * (prefix_tokens * read_mult + variable_tokens)
    ) * base_price / 1_000_000
    return daily_cost * days
```

An example with round numbers: a support assistant on Claude Sonnet 4.5 ($3/MTok input) with a 3,500-token stable prefix (system + tools + examples), 500 variable tokens per request, and 10,000 requests a day. Uncached: about $3,600 a month in input alone. With a 90% hit rate and the 5-minute TTL: about $700. The same traffic on Haiku 4.5 scales the bill down proportionally, and if the flow tolerates asynchronous processing, the Batch API brings cache reads down to 5% of the base price.

Three conclusions that always fall out of this math:

- **Hit rate dominates.** Going from 50% to 90% hits saves more than any other change, including TTL choice. That is why this resource keeps insisting on prompt ordering and deterministic serialization.
- **The right TTL depends on traffic spacing, not prompt size.** Requests every <5 minutes: short TTL. Every 5–60 minutes: 1-hour TTL (Anthropic) or 24h retention (OpenAI). More spread out: only 24-hour retention saves you, or simply do not cache.
- **Small prefixes do not pay.** With 1,200 prefix tokens and 2,000 variable ones, the theoretical maximum saving is a third of the bill. With 30,000 prefix tokens and 500 variable, it is almost all of it. Prioritize caching where the prefix dominates.

Multi-tenant, fan-out, and cache-aware routing

The hit rate you measure in development with a single instance rarely survives real deployment. Three phenomena erode it:

- **Per-tenant prefixes.** If each customer has its own system prompt (custom instructions, their catalog, their tone), each tenant is an independent cache. Design the prompt in layers: first the part common to all tenants (the longest), then the per-tenant block, and the variable content last. The common part is shared and only the tenant layer is written per customer.
- **Concurrent fan-out.** If you fire twenty parallel requests against the same cold prefix, none of them sees the others' cache and you pay twenty writes. The correct pattern is **cache warming**: one short initial request (even with `max_tokens=1`) that writes the prefix, then the fan-out, which now reads. In batch pipelines this saves serious money.
- **Load balancers that spread traffic "well".** A round-robin balancer over multiple API keys or multiple gateway replicas scatters same-prefix requests across different destinations. On OpenAI, `prompt_cache_key` mitigates this on the provider side; in your own gateway, use consistent hashing over a prefix identifier (for example, the hash of the system prompt) so the same agent always lands on the same route.

RAG and prompt caching: friends with conditions

RAG looks like caching's natural enemy: every query retrieves different chunks, and that variable content breaks the prefix. But with three design decisions they coexist well:

- **Separate the stable from the retrieved.** Instructions, response format, few-shot examples, and tool definitions go before the first retrieved chunk, with their breakpoint. That way the instructional prefix (often thousands of tokens) always hits even as documents change.

- **Order chunks deterministically.** If two queries retrieve the same documents, serialize them identically every time (sorted by ID, not by score, which fluctuates). Two requests with the same chunks in the same order also share that part of the prefix. With score ordering, never.
- **In conversational RAG, retrieve once per session when you can.** If the user will ask five questions about the same document, injecting the document once at the start of the conversation (and caching it) is cheaper and faster than retrieving and re-injecting chunks each turn, which invalidates the cached history.

And one alternative that long context has made viable: for medium corpora (up to a few hundred thousand tokens) and concentrated traffic, putting *the entire corpus* in the prompt and caching it (1-hour TTL on Anthropic, 24h retention on OpenAI, or an explicit cache on Gemini) can be simpler, cheaper, and higher quality than a full RAG pipeline with its embeddings, indexes, and reranking infrastructure. Run the numbers with the cost function above before dismissing it.

Agents: compaction, subagents, and tools

Agentic loops are where caching saves the most and where it is easiest to break by accident.

- **Tool results are part of the prefix.** Every `tool_result` that enters the history is cached on the next turn. A 30-step agent with long search results accumulates tens of thousands of tokens which, cached properly, are paid once at write price and afterwards at 10%.
- **Compaction has a price — schedule it.** Summarizing or truncating the history rewrites the prefix and throws away the whole cache. Do not compact "when needed" in the middle of a run of turns: do it at planned cut points (for example, when crossing 70% of the context window), accept that one full-price request, and re-cache the compacted history immediately.
- **Subagents: share the common prefix.** If an orchestrator spawns subagents with partially common instructions, structure the prompts so the shared part is a byte-identical prefix in all of them, with the specialization afterwards. Ten subagents with 4,000 well-ordered common tokens are one write and nine reads.
- **Beware of features that edit the context.** Any mechanism that deletes old tool results or rewrites messages (context editing, automatic history pruning) invalidates the cache from the edited point on. Sometimes it is still worth it (shorter context = fewer tokens on every future turn), but decide with the metrics in front of you, not by default.

Self-hosted: vLLM, SGLang, and prefix caching on your own infrastructure

If you serve open models, prefix caching is not the provider's job: it is yours. The two references:

- **vLLM** implements Automatic Prefix Caching (APC): it slices the KV cache into blocks identified by the prefix hash and reuses blocks across requests, with LRU eviction when memory gets tight. It is enabled by launching the server with `--enable-prefix-caching` and requires no client changes. The server metrics expose the prefix hit rate: watch it the same way you would watch a provider's.
- **SGLang** goes further with RadixAttention: it organizes prefixes into a shared radix tree, so many conversations branching from the same system prompt automatically share the common part without

duplicating it in memory. For workloads with heavy common prefixes (agents, large-scale few-shot, evaluations) the throughput difference is substantial.

The infrastructure lesson is the same as with providers: routing decides the hit rate. With several replicas, a content-blind load balancer spreads identical prefixes across different GPUs and each one recomputes the same thing. Modern inference gateways offer prefix-aware routing: they route by a hash of the start of the prompt so the same prefix always lands on the same replica. If you operate at scale, that piece is worth more than any fine-tuning of the engine.

Security and privacy

- **Per-organization isolation.** On all three major providers the cache is isolated at the organization level: no one outside your account can hit your prefixes. Matching also requires the complete exact prefix: there is no "partial hit" that could reveal content to someone who does not already have it.
- **Extended retention stores tensors, not plain text.** OpenAI's 24-hour mode offloads encrypted KV tensors to GPU-local storage and deletes them on expiry; with Zero Data Retention enabled, extended retention is not used. If you have contractual no-persistence commitments, document which mode you use.
- **Do not put per-user secrets in shared prefixes.** An API token or personal data inside the cached block shared across requests is a design smell: besides fragmenting the cache, it mixes data scopes. Per-user content goes at the end, outside the common prefix.
- **Caching does not change response quality or content.** Reusing the KV cache produces exactly the same computation as reprocessing the prefix. There is no quality trade-off: only cost, latency, and architecture.

Observability: dashboards and alerts

The typical cache regression is not an incident: it is an innocent deploy that adds one dynamic field to the top of the system prompt and multiplies the bill by five without anything "failing". You detect it with metrics, not error logs. Instrument every call:

```
from prometheus_client import Counter

TOKENS = Counter(
    "llm_input_tokens_total",
    "Input tokens by cache type",
    ["model", "kind"],  # kind: read | written | direct
)

def record(model: str, usage) -> None:
    TOKENS.labels(model, "read").inc(
        getattr(usage, "cache_read_input_tokens", 0) or 0)
    TOKENS.labels(model, "written").inc(
        getattr(usage, "cache_creation_input_tokens", 0) or 0)
    TOKENS.labels(model, "direct").inc(usage.input_tokens)
```

With that counter, the hit rate is a PromQL query, and the alert that actually protects your bill is a comparison with the previous week:

```
# Hit rate by model (15-minute window)
sum by (model) (rate(llm_input_tokens_total{kind="read"}[15m]))
/
sum by (model) (rate(llm_input_tokens_total[15m]))

# Alert: hit rate dropped more than 20 points after a deploy
(query_above) < (query_above offset 1w) - 0.20
```

Add the hit rate to the dashboard the team checks after every deploy, next to errors and latency. A sharp drop after a deploy is almost always a prompt change that broke the prefix.

A smoke test in CI

The cheap companion to metrics: a test that fails the pipeline if someone breaks the cacheability of your main prompt. Two identical calls; the second must read from cache:

```
import pytest

def test_main_prompt_is_cacheable(real_client):
    request = build_production_request("ping")

    first = real_client.messages.create(**request)
    second = real_client.messages.create(**request)

    reads = second.usage.cache_read_input_tokens
    assert reads > 0, (
        "Second call did not read from cache: "
        "dynamic field at the top of the prompt? "
        "prefix below the cacheable minimum?"
    )
```

Run it in CI with the real production prompt (built by the same code that serves traffic, not a copy). It is the cost equivalent of a deployment smoke test: cheap, binary, and it fires exactly when someone introduces the regression, not weeks later on the invoice.

Case study: from 12% to 89% hits

To ground everything above, the anatomy of a typical optimization. An internal technical-support assistant: 4,200 tokens of system + tools, an average history of 6 turns, 18,000 daily requests on Sonnet 4.5, initial hit rate of 12%.

- **Diagnosis.** The usage breakdown showed constant writes and almost no reads. Reviewing the prompt: the first line of the system included `Current date: {timestamp}` with seconds, and the tools were serialized from a dict with no fixed order.
- **Fixes, by impact.** (1) The timestamp moved down into the user message and was truncated to the date (no time): the prefix stopped changing per request. (2) Tool serialization frozen with sorted keys. (3) Breakpoint moved to the last message to cache the full history. (4) 1-hour TTL for the system+tools block, because part of the traffic arrived spaced out.

- **Result.** Hit rate at 89%, input cost 71% lower, and the average time-to-first-token dropped from 3.1s to 1.4s on long prompts, because cached prefill also cuts latency. No model change, no quality loss: just ordering and prefix discipline.

Extended checklist

- Start with automatic caching (Anthropic) or default caching (OpenAI, Gemini); move to explicit breakpoints only when you have a concrete reason.
- Log the full usage breakdown (reads, 5m writes, 1h writes, direct) and graph the hit rate per model and per agent.
- Alert on hit-rate drops compared to the previous week, not on absolute thresholds.
- In multi-tenant setups, structure the prompt in layers (common → tenant → variable) and use consistent hashing or `prompt_cache_key` for routing.
- Warm the cache before a concurrent fan-out with one short request.
- In RAG, cache the instructions, order chunks deterministically, and consider corpus-in-prompt for medium corpora with concentrated traffic.
- Schedule history compaction at planned cut points and re-cache immediately.
- Choose TTL by traffic spacing: <5 min → 5m; 5–60 min → 1h or 24h retention; more → 24h or do not cache.
- Run the numbers with the cost function before enabling explicit caches or expensive TTLs.
- A smoke test in CI that verifies the second identical call reads from cache.

Frequently asked questions

Does caching affect response quality? No. Reusing the KV cache is mathematically identical to reprocessing the prefix. Responses may vary between calls due to sampling, but not because of the cache.

Can I cache images and PDFs? Yes: image and document blocks are part of the prefix and cache the same as text. Keep in mind that changing a single pixel or re-encoding the file produces different bytes and breaks the hit.

What happens if two concurrent requests write the same prefix? Both pay for the write; one of the entries prevails. It is harmless but expensive at scale: use the warming pattern before fan-out.

Does streaming change anything? Not for caching: hits are determined at prefill, before the first emitted token. In fact, caching's latency benefit shows up mostly in the time-to-first-token of streamed responses.

Do different users' conversations share a cache? Only byte-identical parts share the cache: the common system and tools do; each user's history does not. That is exactly what you want: the common part is shared, the private part is not.

Why is my hit rate good locally and bad in production? Almost always: routing (several replicas or keys spreading the same prefix), concurrency (duplicate cold writes), or real traffic more spaced out than the TTL. All three have their own section above.

Do framework "system prompts" break the cache? They can. Frameworks that inject variable metadata (run IDs, dates) at the top of the assembled prompt are a classic source of regressions. Inspect the final prompt that goes over the wire, not the one you think you send.

Is it worth it for small prompts? Below the cacheable minimum (1,024–4,096 tokens depending on model and provider) nothing is cached. Just above it, savings are proportional to the prefix's share: with prefixes under ~2,000 tokens you will rarely notice the difference on the bill, though latency does improve somewhat.

How does caching interact with structured output and JSON mode? Structured-output schemas (response_format, output schemas) are part of the request configuration: changing them invalidates the prefix just like changing tools. Keep one stable schema version per endpoint and version it explicitly; if you need per-case variants, treat them as separate agents with their own cache and their own routing key.

Conclusion of the extended edition

Prompt caching started as a billing trick and in 2026 it is an architecture decision: it shapes how you order prompts, how you route traffic, how you design agents and RAG, and even whether you need RAG. The good news is that almost all the value is captured with cheap discipline: a stable prefix, frozen serialization, a TTL matched to your traffic, metrics from day one, and a smoke test guarding the door. The rest — cache-aware routing, explicit caches, corpus-in-prompt — are optimizations you will know how to justify with numbers when volume demands them.

Systematic debugging: when the hit rate does not add up

Sooner or later you will look at the dashboard and the hit rate will not be what you expected. Instead of poking at things randomly, follow a fixed order. Each step rules out an entire family of causes:

- **1. Capture the real prompt that goes over the wire.** Not the template, not the one you think you build: the final payload. A local proxy, an SDK interceptor, or a plain log of the serialized JSON. Most mysteries are solved here: a middleware that adds a context header, a framework that injects the date, a library that reorders keys.
- **2. Diff two consecutive payloads byte by byte.** Save two requests that should share a prefix and run a textual diff. The first differing byte marks where your prefix dies. Everything before that point is cacheable; everything after is not. If the first diff shows up on line 3 of the system prompt, you know what to move to the end.
- **3. Verify the cacheable minimum.** Count the tokens of the stable prefix (the provider's token-counting API, not a character-based estimate). If it is below the model's minimum, there is nothing to debug: either grow the prefix (for example, by moving few-shot examples into the system) or accept that this endpoint does not cache.
- **4. Check the usage breakdown, not just the hit rate.** Constant writes with no reads: the prefix changes on every request (go back to step 2). Writes and reads alternating: the TTL expires between requests (traffic more spaced out than the TTL) or routing spreads traffic across different caches. Everything at zero: the breakpoint is misplaced or below the minimum.

- **5. Isolate the routing.** Replay the traffic from a single process, serially, against a single destination. If the hit rate soars in the experiment and sinks in production, the problem is distribution (replicas, multiple keys, concurrent fan-out), not content.

This order matters because symptoms are deceptive: a 40% hit rate can be a perfect prompt with bad routing, or a broken prompt with perfect routing, and the fixes look nothing alike.

Quick comparison: choosing a strategy per provider

The three providers converge on the idea (exact prefixes, heavily discounted reads) but differ in the control they give you and where the traps are:

- **Anthropic** gives you the finest control: explicit breakpoints with per-block TTLs, automatic caching if you would rather not think about it, a usage breakdown by write type, and stacking with the Batch API. In exchange, writes carry a surcharge: caching traffic that never comes back costs money. It is the clearest mental model for complex agents with long histories.
- **OpenAI** does everything automatically with no write surcharge, so enabling it has no opportunity cost: the worst case is not saving. Control is exercised from the sides, with `prompt_cache_key` for routing and `prompt_cache_retention` for the time window. The classic trap is routing under heavy traffic with similar prefixes.
- **Gemini** splits the two use cases: implicit covers general traffic with zero effort, and explicit is in practice a different product — context rental by the hour — ideal for "many questions about one enormous context". The trap is forgetting to delete explicit caches you no longer use: storage bills for as long as they exist.

If you build multi-provider, design for the common denominator: a byte-stable prefix and dynamic content at the end perform optimally on all three. Provider-specific tuning (breakpoints, routing keys, explicit caches) can live behind your provider-abstraction layer.

Latency: the second dividend

The cost savings are the headline, but in many applications the decisive benefit of caching is latency. Prefill is proportional to prompt length: with 50,000 tokens of context, time-to-first-token can exceed several seconds, and that cost is paid on every turn of the conversation. With the prefix cached, the model only prefills the new tokens.

Practical consequences:

- **Measure TTFT separately for hits and misses.** A single histogram mixes two different distributions and tells you nothing. Label your latency metric with `cache_hit=true/false` and compare: the difference is your business case for investing in hit rate.
- **Caching makes richer prompts viable.** Without a cache, every few-shot example you add makes all requests more expensive and slower. With a cache, the marginal cost of an 8,000-token system versus a 2,000-token one is nearly zero from the second request onward. Many teams cut prompt quality to save money; with good caching, that pressure disappears.

- **In conversational UX, the hit is noticeable.** The difference between 3 seconds and 1 second to the first token is the difference between "it's thinking" and "it answers". If your product competes on perceived speed, hit rate is a product metric, not just a cost metric.

Migration: enabling caching in an existing application

Turning on caching in an app already in production without breaking anything has a sensible order:

- **Week 0: measure without touching.** Add usage-breakdown logging to every call. On OpenAI and Gemini, caching is already on by default: discover your current accidental hit rate. It is usually revealing (and low).
- **Week 1: audit and reorder.** With real payloads in front of you, identify dynamic content in early positions and move it to the end. Freeze the serialization of tools and examples (key order, whitespace). This step requires no cache configuration changes and usually doubles or triples the hit rate on its own.
- **Week 2: enable explicit control where it applies.** On Anthropic, add automatic caching (or breakpoints if you have a reason). On OpenAI, add a per-agent `prompt_cache_key` and evaluate 24h retention for spaced-out traffic. On Gemini, evaluate explicit caching only for giant reused contexts.
- **Week 3: protect the gains.** A smoke test in CI, a hit-rate-drop alert, and one line in the PR template: "does this change touch the system prompt or the tools? → review cache impact". Cache regressions are introduced by well-meaning colleagues in PRs nobody connected to costs.

The most common migration mistake is starting at step 2 (touching configuration) without step 0 (measuring): without a baseline you cannot prove the improvement, and without real payloads you do not know what to reorder.

Anti-patterns we keep seeing

To close, the blacklist. If you recognize yourself in any of these, it is Monday's first task:

- **The "personalized" prompt that personalizes nothing.** Interpolating the user's name into the first sentence of the system ("You are Marta's assistant") as cosmetic courtesy, at the cost of one cache per user. Real personalization goes in the message, not the prefix.
- **The prompt A/B test with no isolation.** Alternating two system variants over the same traffic halves the hit rate for both. If you run prompt experiments, route each variant stickily (the same user always to the same variant) and measure cost per variant.
- **Regenerating tools per request.** Building tool definitions dynamically (filtering by user permissions, for example) produces different prefixes per user. If permission sets are few, generate one stable variant per role; if they are many, consider whether the extra tools can ship anyway and be gated at execution time.
- **Caching with a 1-hour TTL "just in case".** Double the write cost only pays for itself if reads arrive after the 5-minute mark. With dense traffic, the short TTL plus the automatic refresh on every read keeps the cache alive anyway, at half the write price.
- **Trusting the provider dashboard as your only metric.** Billing consoles aggregate by day and by key: they confirm the trend, they do not catch this morning's deploy regression. The metric in your own telemetry, at minute resolution and labeled by deploy version, is the one that warns you in time.

Nothing on this list is exotic: they are reasonable decisions made without the cache in mind. That is the moral of the whole guide. Prompt caching does not ask for heroics; it asks for the prefix to be a stable contract and for someone to be measuring it.

Cost governance: from hit rate to budget

In teams running several LLM-powered features, well-measured caching enables something more valuable than one-off savings: cost attribution. If you tag every call with the feature that originates it (an internal header, a field in your telemetry), you can build the real per-feature cost with caching already discounted, and the conversations change tone: instead of "the AI bill went up 40%", you discuss "autocomplete costs $0.0004 per invocation and document summarization $0.09 — which one do we optimize?".

Three concrete practices that come almost for free once you have cache metrics:

- **Effective cost per request as an economic SLO.** Define a target cost per invocation for each feature (direct tokens + writes + reads, at real prices) and alert when it drifts. Prompt regressions, cache regressions, and model verbosity regressions all show up in that single metric.
- **A write budget.** Cache writes are an investment: they only pay back if reads arrive. A persistently low read/write ratio (<2) on a feature signals a badly sized cache: a TTL too short for its traffic, fragmented prefixes, or traffic too sparse to cache at all.
- **A monthly TTL review.** Traffic patterns change with the product: a feature that went from continuous to sporadic use deserves to move from a short TTL to long retention (or to no caching). Ten minutes a month looking at the per-feature breakdown avoids paying for writes that are never read.

None of this requires new tooling: it is the same usage metrics from this guide, aggregated with one more label. The difference is turning them into a team routine instead of a dashboard curiosity.

Quick glossary

- **Prefill:** the model's initial pass over all input tokens, before generating the first output token. It is what caching avoids repeating.
- **KV cache:** the key-value tensors that prefill produces per layer and per token; the object the provider stores and reuses.
- **Prefix:** the first N tokens of the request, compared byte by byte from the start. The unit of matching for caching.
- **Breakpoint:** an explicit marker (Anthropic `cache_control`) delimiting how far to cache.
- **TTL:** the cache entry's time to live since its last use; reads refresh it.
- **Hit rate:** the fraction of input tokens served from cache over the total. The metric that summarizes your entire strategy.
- **Cache warming:** a deliberate initial request that writes the prefix to cache before a concurrent fan-out.
- **Prefix-aware routing:** directing same-prefix requests to the same server or replica to maximize reuse.