Facsímil 06

Construir y operar

Capítulo 05

Routing, fallback y presupuestos por tarea

686f6c61 · 2026

Capítulo 05: Routing, fallback y presupuestos por tarea

Qué deberías poder hacer al terminar

En el capítulo 02 convertimos una petición en una run con estado y contrato. En el capítulo 03 vimos que servir modelos significa convivir con memoria, colas, prefill, decode y capacidad real. En el capítulo 04 aprendimos a observar una run para reconstruir qué ocurrió.

Ahora toca decidir **por dónde debe ir cada run antes de ejecutarla**.

No todas las tareas merecen el mismo modelo, la misma latencia, el mismo coste ni la misma estrategia de recuperación. Una pregunta corta de soporte no debería consumir el mismo presupuesto que un análisis legal largo. Una extracción JSON estricta no debería ir por una ruta que no soporta salida estructurada. Una tarea interactiva no debería quedarse esperando detrás de un lote nocturno. Y una dependencia lenta no debería provocar diez reintentos que empeoran el sistema.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Diseñar un router operativo.	Separas reglas, señales, catálogo de rutas, presupuesto y trazabilidad.
Definir presupuestos por tarea.	Pones límites de latencia, coste, tokens, retries y salida esperada.
Explicar fallback sin vender atajos.	Distingues degradar, reintentar, cambiar proveedor, cambiar modelo o parar con estado claro.
Calcular coste esperado de una ruta.	Tienes en cuenta entrada, salida, probabilidad de reintento y aceptación final.
Evitar tormentas de reintentos.	Usas timeout, backoff, jitter, circuit breaker y límites de concurrencia.
Instrumentar la decisión.	Registras ruta elegida, alternativas descartadas, motivo y presupuesto consumido.

La idea central: **un sistema serio no llama al modelo “a ver qué pasa”; decide una ruta bajo contrato, presupuesto y señales observables**.

La petición que parece igual pero no lo es

Imagina dos usuarios escribiendo “resume este documento”. La frase parece la misma, pero las condiciones no lo son.

En el primer caso, el documento son cinco párrafos de una política interna y el usuario quiere una respuesta rápida. En el segundo, el documento son setenta páginas, la respuesta debe citar fuentes, el contrato exige JSON, el coste máximo es bajo y la tarea no es interactiva. Si mandamos ambas por la misma ruta, estamos fingiendo que el sistema no entiende su propio trabajo.

El router es la pieza que evita esa ficción. Mira la tarea, el contrato, el presupuesto, la salud de las rutas, la capacidad disponible, la sensibilidad de los datos, el histórico de calidad y decide: local o cloud, modelo pequeño o grande, streaming o batch, RAG o no RAG, salida breve o larga, fallback permitido o no.

Qué no es routing

Routing no es “si falla A, prueba B”. Eso es solo una parte pequeña, y a veces peligrosa, de la historia.

Tampoco es elegir siempre el modelo más potente. El modelo más potente puede ser demasiado caro, lento, innecesario o incompatible con el contrato de salida. En operación, “mejor” no significa “más grande”; significa **suficiente para esta tarea bajo estas restricciones**.

Y tampoco es esconder la decisión dentro de un prompt: “elige tú el mejor modelo”. Algunas decisiones pueden usar un clasificador o un modelo auxiliar, pero la política operativa no debería quedar enterrada en texto. Si cambia el presupuesto, si sube la cola, si una ruta no soporta JSON o si el usuario pertenece a un tenant con región obligatoria, eso debe vivir en una capa explícita.

CONFUSIÓN	QUÉ FALTA
“Router es fallback”	Falta selección inicial, política, coste, capacidad y trazabilidad.
“Uso siempre el modelo grande”	Falta distinguir calidad suficiente de exceso de coste.
“Reintento hasta que salga”	Falta presupuesto, backoff, límite y motivo de parada.
“Si el proveedor va lento, pruebo otro”	Falta saber si el segundo cumple contrato, datos, región y parámetros.
“La ruta la decide el prompt”	Falta una política auditable y versionada.

Qué sí es un router operativo

Un router es, formalmente, una **función de decisión** (un dispatcher) que mapea las señales de una petición a una ruta:

$$\rho(t, c, b, h, o) \to r$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
ρ	Función de routing.	Código, política o servicio que elige ruta.
t	Tipo de tarea.	`support_summary`, `legal_rag`, `json_extract`, `batch_labeling`.
c	Contrato de salida y capacidades requeridas.	JSON estricto, tools, visión, RAG, región, streaming.
b	Presupuesto disponible.	5 segundos, 0,08 EUR, 8000 tokens, 1 retry.
h	Señales de salud y capacidad.	p95, cola, tasa de timeouts, coste p95, proveedor disponible.
o	Observabilidad histórica y evaluación.	aceptación, calidad por ruta, fallos de contrato, golden traces.
r	Ruta elegida.	`local_qwen_fast`, `cloud_frontier_json`, `rag_batch_worker`.

La función puede ser simple o compleja. En un producto pequeño puede ser una tabla YAML con reglas. En un producto grande puede ser un servicio con feature flags, estadísticas por tenant, circuit breakers, evaluaciones offline, rutas canary y aprendizaje sobre resultados. Lo importante no es que sea sofisticado: es que sea **explícito**.

Fecha de corte del estado del arte

Fecha de corte: 28 de mayo de 2026.
Fuentes consultadas: Google SRE sobre sobrecarga y monitorización, AWS Builders Library sobre timeouts, retries, backoff y jitter, OpenRouter Provider Routing, Prompt Caching y Latency and Performance, LiteLLM Router, OpenAI Rate Limits, Latency Optimization, Prompt Caching, Batch API y Flex Processing, Anthropic Rate Limits, Prompt Caching y Cache Diagnostics, Prometheus Metric Naming y OpenTelemetry Semantic Conventions for Generative AI Systems.

Lo estable es el patrón de ingeniería: presupuestos, colas, retries limitados, backoff, jitter, degradación controlada, health signals, circuit breakers, observabilidad y SLOs. Lo cambiante son modelos disponibles, proveedores, precios, límites de cuota, parámetros soportados y sintaxis concreta de routers comerciales.

Google SRE describe la sobrecarga como un problema que exige decidir qué trabajo aceptar, retrasar o rechazar para proteger el servicio.[1] AWS insiste en que timeouts, retries, backoff y jitter deben diseñarse juntos; reintentar sin control puede amplificar el problema que intentas resolver.[2]

En herramientas de IA, OpenRouter documenta selección de proveedor por precio, latencia, throughput, fallbacks, requisitos de parámetros y políticas de datos.[3] LiteLLM documenta un router para balanceo, fallbacks, retries, cooldowns y selección entre despliegues.[4] OpenAI y Anthropic documentan límites de tasa y cuota que obligan a tratar capacidad y presupuesto como parte de la integración, no como detalle posterior.[5][6]

La política de ruta por dentro

Un router maduro no decide con una sola señal. Combina restricciones duras y preferencias.

Primero filtra lo que no cumple:

RESTRICCIÓN DURA	PREGUNTA	EJEMPLO
Contrato	¿La ruta soporta lo que pide la salida?	JSON estricto, tool calling, visión, streaming.
Presupuesto	¿Cabe en coste, tiempo y tokens?	`max_cost_eur <= 0.08`, `max_latency_ms <= 6000`.
Datos	¿La ruta cumple región, retención y política interna?	Local, UE, no almacenar contenido, solo IDs.
Capacidad	¿La ruta puede aceptar trabajo ahora?	Cola, cuota, workers, rate limit, cooldown.
Compatibilidad	¿Soporta los parámetros necesarios?	`temperature`, `response_format`, `max_output_tokens`.

Después ordena lo que sí cumple:

PREFERENCIA	QUÉ OPTIMIZA	CUÁNDO DOMINA
Menor latencia	Tiempo hasta respuesta útil.	Chat interactivo, copilotos, soporte.
Menor coste	Coste por run aceptada.	Clasificación masiva, tareas de bajo margen.
Mayor calidad	Precisión, razonamiento, formato, evaluación.	Tareas complejas, legales, técnicas o de decisión.
Mayor estabilidad	Menos variación p95/p99.	Productos con SLO estricto.
Menor exposición de datos	Menos salida de datos del entorno controlado.	Documentos internos o datos sensibles.

Para elegir ruta podemos puntuar cada candidata con una función de valor aditiva, la decisión multicriterio de siempre: sumar lo que aporta y restar lo que cuesta, cada término con su peso.[7]

$$S(r) = w_q Q(r) - w_c C(r) - w_l L(r) - w_e E(r) - w_s \sigma(r)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$S(r)$	Puntuación de la ruta r.	0,71 para `local_fast`; 0,83 para `cloud_json`.
$Q(r)$	Calidad esperada normalizada.	0,92 si pasa evals de extracción.
$C(r)$	Coste esperado normalizado.	0,30 si cuesta poco, 0,90 si es caro.
$L(r)$	Latencia esperada normalizada.	p95 de 2,5 s frente a un límite de 6 s.
$E(r)$	Penalización por exposición de datos o restricciones.	0 si todo queda local; 0,6 si sale a cloud sin necesidad.
$\sigma(r)$	Variabilidad o cola de cola larga.	Penaliza rutas con p99 malo aunque p50 parezca bien.
w_q, w_c, w_l, w_e, w_s	Pesos de decisión.	En soporte pesa latencia; en extracción pesa contrato.

Esto no obliga a convertir el router en una fórmula rígida. La fórmula sirve para una idea: una ruta no se elige solo porque “responde bien”, sino porque equilibra calidad, coste, latencia, política y estabilidad.

Catálogo de rutas: la tabla que sostiene al router

Un router sin catálogo acaba siendo un `if` enorme. Al principio parece suficiente, pero pronto nadie sabe qué rutas existen, qué contrato soportan, qué coste tienen, qué dueño las mantiene o qué limitación ocultan.

El catálogo de rutas es un artefacto versionado. Puede vivir en YAML, base de datos, configuración de plataforma o servicio interno, pero debe responder preguntas concretas:

PREGUNTA	CAMPO DEL CATÁLOGO
¿Qué ruta puedo usar para esta tarea?	`allowed_tasks`, `capabilities`, `contracts`.
¿Qué ruta no debo usar con estos datos?	`data_policy`, `region`, `retention`.
¿Cuánto cuesta antes de ejecutarla?	`price.input_1k`, `price.output_1k`, `tool_costs`.
¿Qué SLO suele cumplir?	`latency.p50`, `latency.p95`, `ttft.p95`, `tpot.p95`.
¿Qué límites tiene ahora mismo?	`rpm`, `tpm`, `concurrency`, `context_window`.
¿Quién responde si se rompe?	`owner`, `runbook`, `dashboard`.
¿Cómo se degrada?	`fallbacks`, `degraded_mode`, `stop_state`.

Una versión mínima:

```yaml
version: route-catalog@2026-05-27
owner: ai-runtime

routes:
  local_fast:
    provider: local
    model: qwen3-8b-instruct
    runtime: vllm
    owner: ai-platform
    allowed_tasks: [fast_triage, support_fast_answer]
    capabilities: [text, json]
    contracts: [triage_v2, short_answer_v1]
    context_window: 32768
    data_policy:
      region: local
      retention: none
      content_logging: ids_only
    price:
      input_1k_eur: 0.0002
      output_1k_eur: 0.0006
    observed:
      latency_p95_ms: 1800
      timeout_ratio_5m: 0.02
      contract_failure_ratio_24h: 0.012
    limits:
      max_concurrency: 24
      max_output_tokens: 700
    fallback:
      chain: [support_brief_static]
      stop_state: capacity_unavailable

  rag_medium_json:
    provider: cloud
    model: medium-json-2026-05
    owner: ai-runtime
    allowed_tasks: [support_summary, policy_qa]
    capabilities: [text, json, rag, citations]
    contracts: [answer_with_sources_v3]
    context_window: 131072
    data_policy:
      region: eu
      retention: none
      content_logging: hash_and_ids
    price:
      input_1k_eur: 0.002
      output_1k_eur: 0.006
    observed:
      latency_p95_ms: 4200
      timeout_ratio_5m: 0.05
      contract_failure_ratio_24h: 0.018
    limits:
      rpm: 600
```

```
      tpm: 1200000
      max_output_tokens: 1200
    fallback:
      chain: [rag_small_brief, batch_review]
      stop_state: no_route_under_budget
```

El catálogo también evita una trampa frecuente: pensar que “modelo” y “ruta” son lo mismo. Una ruta incluye modelo, proveedor, runtime, región, parámetros permitidos, contrato, límites, observabilidad, owner y fallback. Dos rutas pueden usar el mismo modelo y comportarse distinto porque cambian proveedor, cuantización, template, latencia, región o versión del gateway.

Presupuesto por tarea

Un presupuesto de tarea es un contrato operativo, un vector de límites. No dice «usa poco»: dice cuánto se puede gastar, esperar y reintentar.

$$B_t = \langle T_{max}, C_{max}, I_{max}, O_{max}, R_{max}, A_{max} \rangle$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
B_t	Presupuesto para la tarea t.	Presupuesto de `support_summary`.
T_{max}	Latencia máxima.	6000 ms.
C_{max}	Coste máximo.	0,08 EUR.
I_{max}	Tokens máximos de entrada.	12.000 tokens.
O_{max}	Tokens máximos de salida.	600 tokens.
R_{max}	Reintentos máximos.	1 retry.
A_{max}	Acciones o llamadas externas máximas.	2 tools, 1 retrieval, 0 escrituras.

Ejemplo de presupuestos:

TAREA	LATENCIA	COSTE	TOKENS	REINTENTOS	RUTA PREFERIDA
`support_fast_answer`	2500 ms	0,02 EUR	4000 entrada / 300 salida	0	modelo pequeño o caché.
`support_summary`	6000 ms	0,08 EUR	12000 entrada / 600 salida	1	RAG + modelo medio.
`legal_rag_review`	30000 ms	0,60 EUR	60000 entrada / 2000 salida	1	modelo fuerte + citas + revisión.
`batch_labeling`	120000 ms	0,01 EUR por item	2000 entrada / 50 salida	2	lote barato, sin streaming.
`json_extract_strict`	8000 ms	0,05 EUR	8000 entrada / 800 salida	1	ruta con salida estructurada fiable.

La clave está en el adjetivo “por tarea”. Un presupuesto global tipo “máximo 20.000 tokens” no dice casi nada. La operación necesita saber si esos tokens pertenecen a una conversación interactiva, a un lote, a una extracción o a un resumen con citas.

Cuotas y rate limits: capacidad externa también es arquitectura

Cuando usas proveedores externos, la capacidad no depende solo de tu código. También depende de cuotas de peticiones, tokens, concurrencia, región, tier, modelo y organización. OpenAI y Anthropic documentan límites de tasa que pueden expresarse en peticiones, tokens u otros contadores según producto, modelo o plan.[89]

Para un router, esos límites no son un error inesperado: son una entrada de decisión.

Para no saturar un proveedor, la regla de capacidad es una desigualdad de tasa: las llegadas por los tokens esperados de cada una no deben superar el límite de la ruta, dejando un margen de seguridad.

$$\lambda_t \cdot E[I_t + O_t] \leq TPM_{ruta} \cdot \alpha$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
λ_t	Runs por minuto de la tarea t.	120 resúmenes por minuto.
$E[I_t + O_t]$	Tokens esperados por run.	7500 entrada + 500 salida.
TPM_{ruta}	Tokens por minuto permitidos en la ruta.	1.200.000 TPM.
α	Margen operativo que no queremos superar.	0,75 para dejar colchón.

Si $120 \cdot 8000 = 960000$ tokens por minuto y la ruta tiene 1.200.000 TPM, parece caber. Pero con $\alpha = 0,75$, el límite operativo sería 900.000 TPM. El router debería empezar a derivar parte del tráfico, pasar una clase a batch o bajar tokens de entrada.

Lo mismo para peticiones por minuto:

$$\lambda_t \leq RPM_{ruta} \cdot \alpha$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
RPM_{ruta}	Peticiones por minuto permitidas.	600 RPM.
λ_t	Llegada esperada.	520 RPM.
α	Margen de seguridad.	0,80.

Con $\alpha = 0,80$, 600 RPM se convierten en 480 RPM operativos. Si llegan 520, no conviene esperar al error de cuota; conviene actuar antes.

SEÑAL	ACCIÓN RAZONABLE DEL ROUTER
`quota_remaining_tpm` baja rápido.	Reducir contexto, cambiar ruta o mandar tareas no urgentes a batch.
`rpm_utilization` supera margen.	Activar backpressure o repartir entre rutas compatibles.
`retry_after_ms` cabe en presupuesto.	Esperar con backoff y jitter.
`retry_after_ms` no cabe.	Cambiar ruta o cerrar con estado claro.
`context_window` insuficiente.	Comprimir, hacer RAG, dividir tarea o rechazar con explicación.

El punto de ingeniería: no esperes a que el proveedor te diga “no puedo”. Tu router debería saber antes si la ruta está cerca del límite.

Coste esperado y coste útil

El coste visible de una llamada no es el coste de producto. Si una ruta falla contrato, requiere reintento o produce una respuesta que se descarta, ha consumido dinero sin producir utilidad.

El coste esperado de una run es la suma de sus partes, con la probabilidad de reintento incluida. Es una esperanza matemática, no un teorema:

$$E[C_{run}] = C_{in} + C_{out} + p_{retry}(C_{in}^{retry} + C_{out}^{retry}) + C_{tool} + C_{retrieval}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$E[C_{run}]$	Coste esperado de una run.	0,047 EUR.
C_{in}	Coste de tokens de entrada.	10.000 tokens de contexto.
C_{out}	Coste de tokens de salida.	500 tokens generados.
p_{retry}	Probabilidad de necesitar reintento.	0,08.
$C_{in}^{retry}, C_{out}^{retry}$	Coste de un intento adicional.	Puede ser menor si recortamos contexto.
C_{tool}	Coste de tools o servicios externos.	Búsqueda, OCR, base vectorial, API externa.
$C_{retrieval}$	Coste de recuperación y reranking.	Embeddings, vector DB, reranker.

Y el coste que de verdad importa es el **coste por tarea aceptada** (cost per successful task): el coste total repartido entre las ejecuciones que pasan el control.

$$C_{util} = \frac{\sum C_{run}}{N_{aceptadas}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
C_{util}	Coste por run aceptada.	0,11 EUR por respuesta útil.
$\sum C_{run}$	Coste total de runs intentadas.	11 EUR.
$N_{aceptadas}$	Runs que pasan contrato y sirven al usuario.	100.

Esta fórmula es incómoda porque revela rutas “baratas” que salen caras. Un modelo pequeño puede costar poco por token, pero si falla más el contrato, reintenta más o exige revisión, el coste útil puede subir.

Tipos de routing que sí aparecen en producción

No hay un único tipo de router. Hay familias.

TIPO	CÓMO DECIDE	SIRVE CUANDO	CUIDADO
Reglas estáticas	`task == X -> ruta Y`.	Producto pequeño, compliance claro, pocas tareas.	Se queda rígido si crece el catálogo.
Routing por contrato	Filtra por soporte de JSON, tools, visión, región o streaming.	La salida tiene requisitos fuertes.	Hay que mantener capacidades por ruta.
Routing por coste	Ordena por coste estimado bajo una calidad mínima.	Tareas masivas o de bajo margen.	Barato no equivale a aceptable.
Routing por latencia	Prefiere p95/p99 menor o cola más corta.	Experiencia interactiva.	Puede elegir rutas más caras.
Routing por calidad	Usa evals, golden traces y resultados históricos.	Tareas donde el error cuesta más que la latencia.	Necesita datasets vivos y versionados.
Routing por salud	Evita rutas con timeouts, cuota agotada o cola alta.	Operación real con proveedores y runtimes cambiantes.	Sin observabilidad, se convierte en intuición.
Routing híbrido	Combina reglas, score, salud, coste y contrato.	Sistemas de IA en producción.	Requiere trazabilidad y ownership.

Lo importante: un router puede empezar simple y evolucionar. Lo peligroso es empezar opaco.

Compatibilidad real entre proveedores

Una API compatible no garantiza comportamiento compatible. Dos rutas pueden aceptar un cuerpo parecido y aun así diferir en lo que importa.

OpenRouter documenta opciones para exigir proveedores que soporten parámetros concretos y para filtrar por políticas de datos, proveedores, cuantización, precio, latencia o throughput.[10] Esa idea es general: antes de fallback o routing cruzado, hay que comprobar compatibilidad.

SUPERFICIE	QUÉ COMPARAR	POR QUÉ IMPORTA
Salida estructurada	JSON mode, schema estricto, tool calling, validación.	Una ruta puede “intentar JSON” y otra garantizar contrato.
Streaming	Formato de eventos, fin de stream, errores parciales.	El cliente puede depender de eventos concretos.
Tools	Schema, tool choice, llamadas paralelas, argumentos.	Cambia cómo el modelo decide usar herramientas.
Contexto	Ventana, tokenizador, coste de entrada, compaction.	El mismo texto no ocupa igual ni cabe igual.
Parámetros	`temperature`, `top_p`, `seed`, `max_tokens`, razonamiento.	El nombre puede existir, pero su efecto no ser idéntico.
Seguridad y datos	Región, retención, logging, entrenamiento con datos.	No todas las rutas sirven para todos los tenants.
Errores	Códigos, `retry_after`, timeouts, cuota.	El gateway debe normalizar estados para el router.
Precios	Input, output, caché, tools, razonamiento, lote.	El coste esperado necesita la factura completa.
Versionado	Alias flotante o versión fija.	Un alias puede cambiar comportamiento sin cambiar tu código.

Por eso el catálogo debería distinguir `supports_json_schema=true` de `usually_returns_json=true`. Lo primero es una capacidad. Lo segundo es una esperanza.

Un ejemplo de matriz de compatibilidad:

RUTA	CAPACIDADES DECLARADAS	ENTORNO	FALLBACK PERMITIDO
`local_fast`	JSON estricto, streaming, sin tools. Contexto 32768.	Local.	Sí, a respuesta breve.
`rag_medium_json`	JSON estricto, tools, streaming, RAG y citas. Contexto 131072.	UE.	Sí, a `rag_small_brief`.
`frontier_review`	JSON estricto, tools, streaming y contexto largo.	Proveedor.	Solo si presupuesto lo permite.
`batch_cheap`	JSON estricto, sin streaming, sin tools. Contexto 32768.	Proveedor.	No interactivo.

La matriz no es burocracia. Es lo que impide que un fallback rompa el contrato.

Fallback no significa “hacer cualquier cosa”

Fallback es una ruta alternativa con contrato. No es una excusa para devolver algo distinto sin avisar.

Hay varios tipos:

TIPO DE FALLBACK	QUÉ CAMBIA	EJEMPLO
Mismo modelo, otro proveedor	Cambia endpoint o región.	Mismo modelo servido por otro proveedor compatible.
Otro modelo equivalente	Cambia modelo manteniendo contrato.	Modelo B que soporta JSON y calidad mínima.
Modelo menor	Reduce coste o latencia.	Respuesta breve, menos contexto, menos razonamiento.
Ruta local	Evita dependencia externa.	Modelo local para resumen simple cuando cloud no conviene.
Ruta batch	Cambia expectativa temporal.	“Lo dejamos procesando y avisamos cuando esté”.
Respuesta parcial	Devuelve lo seguro y marca lo pendiente.	“Puedo resumir, pero no validar citas ahora”.
Parada explícita	No ejecuta más.	`budget_exhausted`, `contract_not_supported`, `capacity_unavailable`.

La regla de oro: **fallback debe ser igual o más conservador que la ruta principal**. Si la ruta principal no puede ejecutar una acción con el contrato correcto, el fallback no debería inventar más capacidad. Debe reducir alcance, pedir confirmación, pasar a batch o cerrar con estado comprensible.

Reintentos, backoff y jitter

Un retry solo tiene sentido si el problema puede desaparecer al volver a intentar. Un error de contrato no se arregla repitiendo igual. Una cuota agotada no se arregla enviando más rápido. Una salida inválida quizá se arregla con un retry si cambiamos instrucción, temperatura o modelo, pero hay que presupuestarlo.

Política mínima:

CASO	RETRY	MOTIVO
Timeout de red puntual	Sí, con backoff y límite.	Puede ser transitorio.
Rate limit	Sí, si `retry_after` cabe en presupuesto.	Si no cabe, cambiar ruta o parar.
JSON inválido	Una vez, con reparación o ruta más fiable.	Repetir igual suele repetir el fallo.
Presupuesto agotado	No.	Ya no hay margen operativo.
Contrato no soportado	No.	La ruta era incorrecta.
Error de permisos	No.	No es un problema temporal.

Backoff exponencial simple:

$$delay_i = \min(D_{max}, D_0 \cdot 2^i) + U(0, J)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$delay_i$	Espera antes del retry i.	400 ms, 850 ms, 1700 ms.
D_0	Espera inicial.	250 ms.
D_{max}	Espera máxima.	3000 ms.
i	Número de retry.	0, 1, 2.
$U(0, J)$	Ruido aleatorio entre 0 y J.	Jitter de 0 a 150 ms.

El jitter parece un detalle menor, pero evita que muchos clientes despierten a la vez y vuelvan a presionar la misma dependencia. En sistemas con IA, donde cada intento puede ser caro, el retry debe gastar presupuesto explícito.

Circuit breaker, cooldown y load shedding

Cuando una ruta empieza a fallar o a degradarse, el router no debería seguir probándola con todo el tráfico. Necesita memoria operativa.

MECANISMO	QUÉ HACE	SEÑAL TÍPICA
Circuit breaker	Cierra temporalmente una ruta con demasiados fallos recientes.	`timeout_ratio_1m > 0.25`.
Cooldown	Espera antes de volver a probar la ruta.	30 segundos, 2 minutos, 10 minutos.
Probe	Envía pocas runs de prueba antes de reabrir.	1% del tráfico o tarea sintética.
Load shedding	Rechaza o retrasa trabajo para proteger el sistema.	Cola superior a umbral o SLO quemándose.
Backpressure	Indica al cliente o cola que reduzca entrada.	`retry_after`, cola batch, límite por tenant.

Esto conecta con Little:

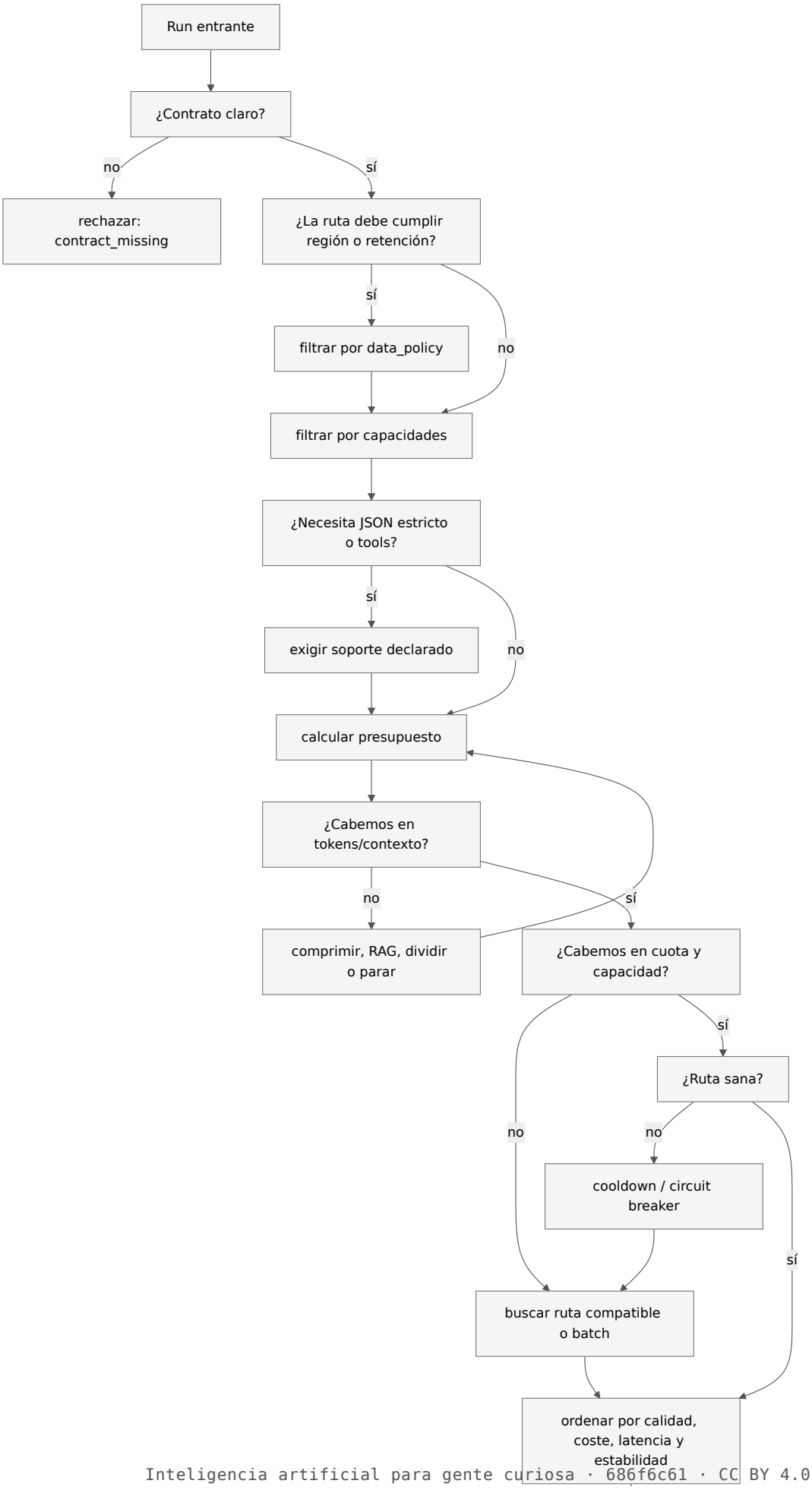
$$L = \lambda W$$

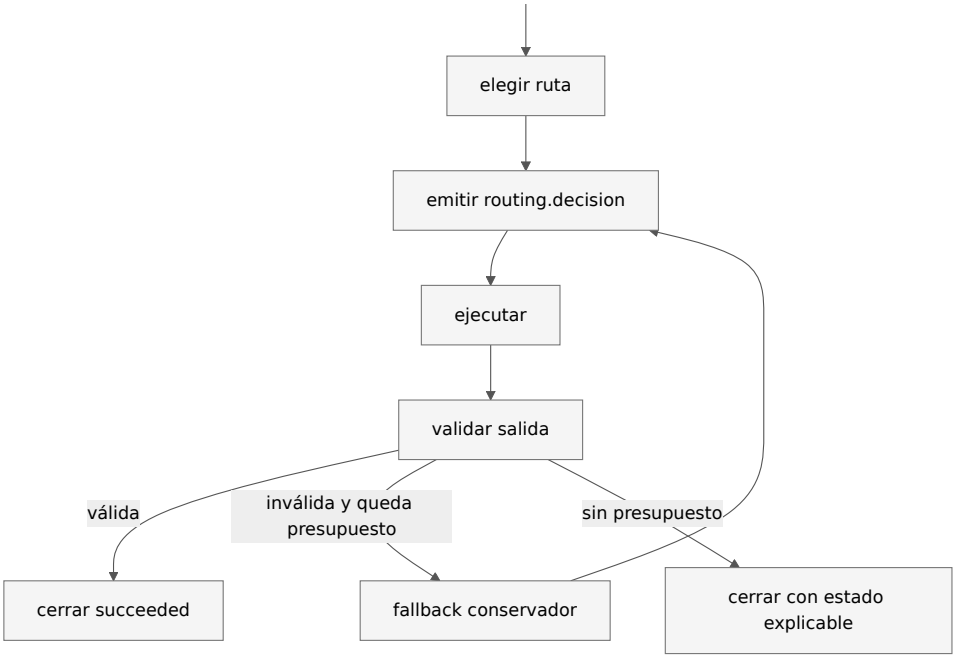
SÍMBOLO	SIGNIFICADO	EJEMPLO
L	Trabajo medio dentro del sistema.	240 runs esperando o ejecutándose.
λ	Tasa de llegada.	40 runs por minuto.
W	Tiempo medio en el sistema.	6 minutos.

Little no es una receta universal; es una alarma conceptual. Si la llegada sube y la capacidad no sube, el tiempo de espera crece. Un router que acepta todo “porque quizá sale” puede destruir el SLO. Dean y Barroso explicaron en *The Tail at Scale* que la cola larga de latencia importa porque el usuario vive los peores percentiles, no la media.[11] Little formuló la relación clásica entre trabajo en sistema, llegada y espera.[12]

Árbol de decisión para elegir ruta

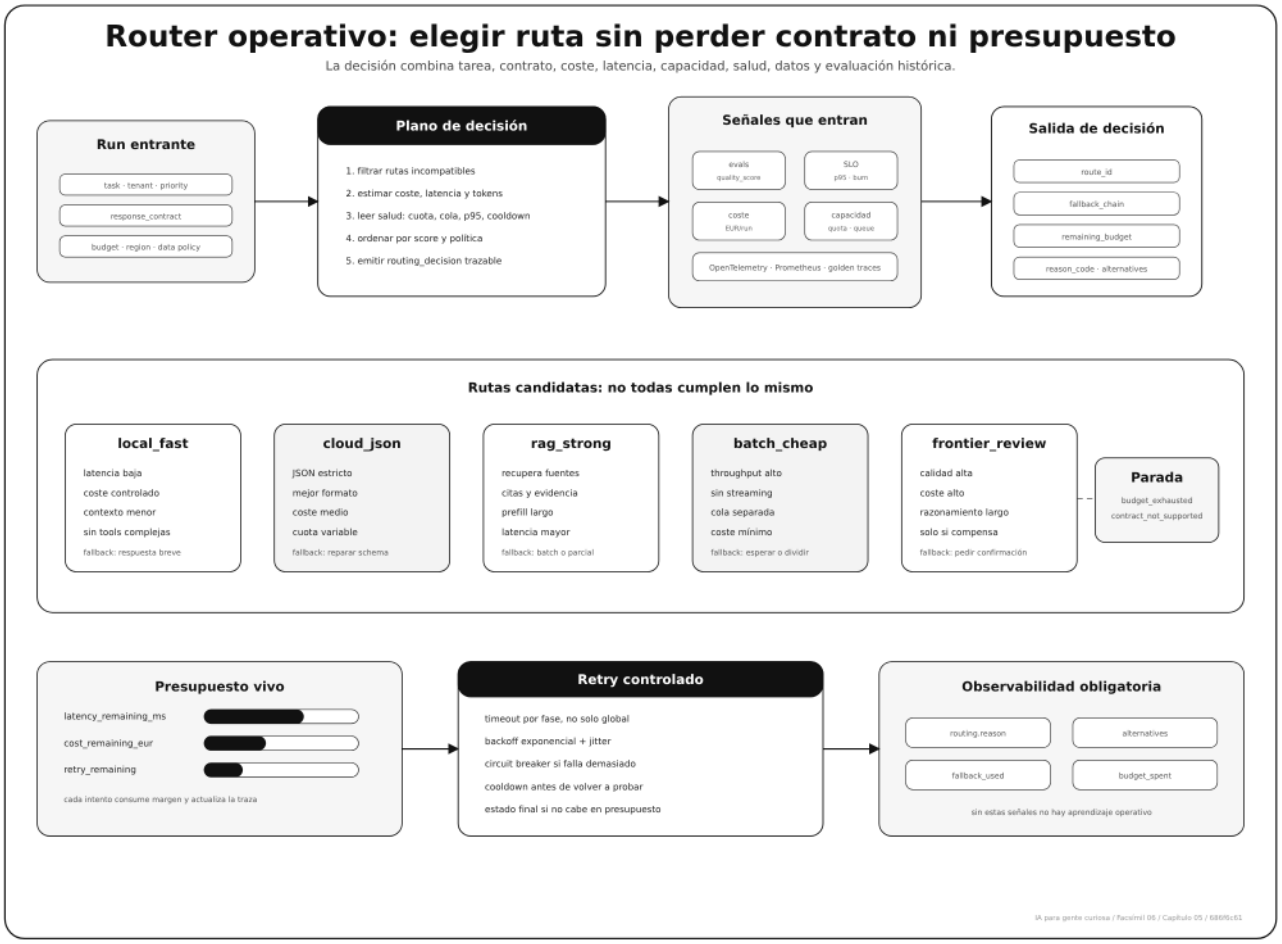
El árbol no sustituye al router, pero ayuda a revisar si la política tiene sentido. Léelo de arriba abajo:


Run entrante
¿Contrato claro?
no
sí
rechazar: contract_missing
¿La ruta debe cumplir región o retención?
sí
no
filtrar por data_policy
filtrar por capacidades
¿Necesita JSON estricto o tools?
sí
no
exigir soporte declarado
calcular presupuesto
¿Cabemos en tokens/contexto?
no
sí
comprimir, RAG, dividir o parar
¿Cabemos en cuota y capacidad?
sí
¿Ruta sana?
no
no
cooldown / circuit breaker
sí
buscar ruta compatible o batch
ordenar por calidad, coste, latencia y estabilidad



Fíjate en dos ramas: primero contrato, después presupuesto. Si el contrato falta, no hay router inteligente; hay adivinanza. Si el presupuesto no cabe, no hay “un intento más”; hay deuda operativa.

Anatomía visual de un router de IA



El dibujo enseña la idea importante: el router no es una bifurcación simple. Es una frontera operativa. Antes de gastar tokens decide si una ruta cumple contrato, si cabe en presupuesto, si está sana y qué alternativa existe si algo no encaja.

Cómo se ve en producción

Una decisión de routing debería quedar registrada como dato estructurado. Algo así:

```
{
  "event": "routing.decision",
  "run_id": "run_20260527_0512",
  "task": "support_summary",
  "selected_route": "rag_medium_json",
  "reason": "meets_contract_under_budget",
  "budget": {
    "max_latency_ms": 6000,
    "max_cost_eur": 0.08,
    "max_retries": 1
  },
  "estimated": {
    "latency_p95_ms": 4200,
    "cost_eur": 0.041,
    "input_tokens": 7200,
    "output_tokens": 480
  },
  "alternatives": [
    {
      "route": "local_fast",
      "rejected_because": "does_not_support_required_citations"
    },
    {
      "route": "frontier_review",
      "rejected_because": "cost_above_task_budget"
    }
  ],
  "fallback_chain": ["rag_medium_json", "rag_small_brief", "batch_review"]
}
```

Esto no está para decorar logs. Sirve para responder preguntas:

PREGUNTA	CAMPO QUE LA RESPONDE
¿Por qué no usamos el modelo local?	`alternatives.rejected_because`.
¿Por qué subió el coste?	`estimated.cost_eur`, ruta seleccionada, tokens y retries.
¿Cuántas veces se activa fallback?	`fallback_chain`, `fallback_used`, `final_state`.
¿Qué ruta incumple p95?	`selected_route` + métricas de latencia.
¿Qué contrato expulsa rutas?	`response_contract` y `does_not_support_*`.

Prometheus recomienda nombres de métricas y labels que permitan entender unidades y agregación sin crear cardinalidad inútil.[13] OpenTelemetry mantiene convenciones semánticas para sistemas generativos, incluyendo atributos de modelo, operación y uso de tokens.[14] En

nuestro router, eso se traduce en no inventar métricas imposibles de mantener: `ai_route_decision_total{task,route,reason}`, sí; `ai_route_decision_total{run_id}`, no.

Herramientas y piezas que se suelen usar

Hay tres caminos frecuentes.

CAMINO	QUÉ USAS	PROS	CONTRAS
Router propio	Código interno, YAML, feature flags, métricas y gateway.	Control total sobre contratos, coste, datos y trazas.	Hay que mantener catálogo, salud, cuotas y fallbacks.
Router de proveedor/a gregador	OpenRouter, gateway cloud o proveedor con múltiples endpoints.	Reduce integración con muchos proveedores; aporta routing por precio, latencia o proveedor.	No sustituye tu política de producto ni tus presupuestos internos.
Router de librería	LiteLLM Router u otra capa compatible.	Acelera balanceo, retries y fallbacks entre despliegues.	Debes revisar semántica de parámetros, errores, streaming, tools y observabilidad.

Mi recomendación para ingeniería: aunque uses un router externo, conserva un **router lógico propio**. Ese router lógico decide tarea, contrato, presupuesto y política. Luego puede delegar la ejecución a OpenRouter, LiteLLM, un proveedor cloud o un runtime local. Si toda la decisión vive fuera de tu sistema, pierdes criterio de producto.

Optimizar el router: qué palancas existen

Optimizar no significa “usar siempre lo más barato”. Tampoco significa “usar siempre lo más rápido”. Optimizar significa mejorar una métrica bajo restricciones: coste por run aceptada, p95 de latencia, TTFT, tasa de contrato válido, tasa de fallback, cuota restante, calidad evaluada o experiencia percibida por el usuario.

La primera regla es elegir el objetivo:

OBJETIVO	OPTIMIZACIÓN CORRECTA	OPTIMIZACIÓN PELIGROSA
Bajar latencia interactiva	Reducir salida, usar modelo menor suficiente, streaming, caché de prefijo, ruta con p95 estable.	Recortar contexto sin medir calidad.
Bajar coste	Batch, flex, prompt caching, modelo menor, menos tokens, menos retries, cachear respuestas deterministas.	Cambiar a ruta barata que sube fallo de contrato.
Mejorar throughput	Colas por clase, batch, rutas con más TPS, bajar saturación, repartir por cuota.	Aumentar concurrencia hasta disparar p99.
Mejorar estabilidad	Circuit breaker, cooldown, margen de cuota, rutas canary, percentiles p90/p99.	Mirar solo p50 o media.
Mantener calidad	Evals por tarea, golden traces, matriz de compatibilidad, fallback conservador.	Cambiar modelo sin dataset de comparación.

OpenAI resume la optimización de latencia en principios bastante prácticos: procesar tokens más rápido, generar menos tokens, usar menos tokens de entrada, hacer menos peticiones, paralelizar, reducir la espera percibida y no usar un LLM cuando una técnica clásica basta.[15] Para un router, eso se traduce en política.

PALANCA	QUÉ CAMBIA	SEÑAL QUE DEBE MEJORAR	RIESGO
Modelo menor suficiente	Ruta más rápida y barata.	TTFT, TPOT, coste.	Baja calidad si no hay evals.
Menos salida	`max_output_tokens`, contrato más compacto, respuesta breve.	Latencia de decode y coste de salida.	Respuesta incompleta.
Menos entrada	RAG más selectivo, limpiar HTML, resumir historial, deduplicar contexto.	Coste de entrada y prefill.	Perder evidencia necesaria.
Prefijo estable	Instrucciones, ejemplos y tools al principio; datos variables al final.	Cache hit rate, TTFT, coste de entrada.	Cache miss silencioso si cambian timestamps o tool order.
Menos llamadas	Unir pasos secuenciales en una respuesta estructurada.	Latencia total y coste de red.	Prompt demasiado complejo.
Paralelizar	Ejecutar pasos independientes a la vez.	Latencia de pared.	Más coste si luego cancelas trabajo.
Streaming	Primer token antes, usuario espera menos.	TTFT y experiencia percibida.	No reduce coste por sí mismo.
Batch	Llevar tareas no urgentes a procesamiento asíncrono.	Coste y cuota síncrona disponible.	No sirve para interacción inmediata.
Flex o baja prioridad	Menor coste a cambio de más espera o disponibilidad variable.	Coste por run no urgente.	Más timeouts o `resource_unavailable`.
Response cache	Reutilizar respuesta exacta si entrada y contrato son repetibles.	Latencia y coste casi cero en hits.	Solo sirve si la respuesta puede repetirse.

La latencia de una run se descompone en sus etapas, siguiendo el modelo de serving con fase de prefill y fase de decode.[16]

$$T_{run} \approx T_{queue} + \frac{I_{uncached}}{R_{prefill}} + \frac{O}{R_{decode}} + T_{tools} + T_{network}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T_{run}	Latencia total aproximada.	4200 ms.
T_{queue}	Espera en cola o proveedor.	300 ms.
$I_{uncached}$	Tokens de entrada que realmente se procesan.	1200 dinámicos + 0 cacheados.
$R_{prefill}$	Velocidad de procesamiento de entrada.	tokens/s en prefill.
O	Tokens de salida generados.	480 tokens.
R_{decode}	Velocidad de generación.	tokens/s en decode.
T_{tools}	Tiempo de tools, RAG, validación o reranking.	650 ms.
$T_{network}$	Red, gateway y serialización.	120 ms.

La fórmula explica por qué reducir salida suele notarse más que recortar un poco de entrada en chats normales: el decode produce tokens de uno en uno. Pero en documentos largos, RAG grande o tools repetidas, el prefill, el caché y la recuperación sí pueden dominar.

Para coste con caché:

$$C_{in} = I_{fresh}P_{in} + I_{cache\_write}P_{write} + I_{cache\_read}P_{read}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
I_{fresh}	Tokens nuevos no cacheados.	pregunta del usuario y chunks dinámicos.
P_{in}	Precio normal de entrada.	precio por 1M tokens.
I_{cache_write}	Tokens que se escriben en caché.	system prompt largo o tools.
P_{write}	Precio de escritura de caché.	puede ser mayor que entrada normal.
I_{cache_read}	Tokens leídos desde caché.	prefijo estable reutilizado.
P_{read}	Precio de lectura de caché.	normalmente menor que entrada normal.

La conclusión: cachear compensa cuando hay prefijos largos y repetidos. No compensa si cada prompt empieza distinto.

Cuánta fiabilidad te da una cadena de fallback. Tener un modelo de respaldo no es una vaga sensación de seguridad: añade fiabilidad de forma calculable. Si la ruta y sus respaldos fallan de manera independiente, la cadena solo falla si fallan todos, y eso es un producto de probabilidades.[17]

$$R_{\text{cadena}} = 1 - \prod_{i=1}^{n} (1 - r_i)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
r_i	Fiabilidad de cada ruta (probabilidad de que responda bien).	0,95 y 0,90.
n	Número de rutas en la cadena de fallback.	2
R_{cadena}	Fiabilidad del conjunto con respaldo.	$1 - 0{,}05 \times 0{,}10 = 0{,}995$.

En palabras: un modelo al 95% con un respaldo al 90% da un 99,5% combinado, siempre que los fallos sean independientes. El matiz es ese «independientes»: si ambos dependen del mismo proveedor o de la misma red, los fallos se correlacionan y la cuenta optimista no se cumple. Un buen fallback usa una vía de fallo distinta, no otra copia del mismo riesgo.

Cómo explorar rutas sin malgastar: UCB1. Si no sabes qué ruta es mejor, tienes un dilema clásico: explotar la que parece buena o explorar las demás por si lo son más. El algoritmo UCB1 lo resuelve con una fórmula que premia tanto el buen rendimiento medio como la incertidumbre por haber probado poco.[18]

$$\text{UCB}_i = \bar{x}_i + \sqrt{\frac{2 \ln n}{n_i}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\bar{x}_i$	Recompensa media observada de la ruta i (calidad por coste).	0,82
n_i	Veces que se ha usado la ruta i.	40
n	Veces totales de routing.	1.000
UCB_i	Puntuación que decide la ruta del próximo caso.	Suma media más bono de exploración.

En palabras: se elige la ruta con mayor puntuación, donde el segundo término es un «bono» que crece si una ruta se ha probado poco. Así el sistema prueba lo justo las opciones dudosas y converge a la buena. Es el reemplazo honesto del «creo que el modelo caro es mejor»: que lo demuestren los datos, sin dejar de explorar.

Cuánto cuestan de verdad los reintentos. Un reintento parece gratis hasta que se cuenta. Si cada intento falla con probabilidad q de forma independiente, el número de intentos hasta acertar sigue una distribución geométrica, y su media es conocida.[19]

$$\mathbb{E}[N] = \frac{1}{1-q}, \qquad \mathbb{E}[C] = \frac{c}{1-q}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q	Probabilidad de que un intento falle.	0,2
$\mathbb{E}[N]$	Número medio de intentos hasta el éxito.	$1/0{,}8 = 1{,}25$.
c	Coste de un intento.	0,03 euros.
$\mathbb{E}[C]$	Coste medio por tarea resuelta, con reintentos.	0,0375 euros.

En palabras: con un 20% de fallos, resolver una tarea cuesta de media un 25% más de lo que sugiere el precio por llamada. Por eso el coste por tarea aceptada y el coste por llamada divergen, y por eso un límite de reintentos no es tacañería: sin tope, una racha de fallos multiplica la factura sin garantizar el éxito.

Cómo respetar un presupuesto sin frenar en seco: el cubo de tokens. Un límite de peticiones por minuto no debería ser un muro que rechaza todo al pasarse. El algoritmo del cubo de tokens permite ráfagas acotadas y un ritmo sostenido: el cubo se rellena a tasa constante y cada petición gasta tokens; si no hay, espera o se rechaza.[20]

$$\text{tokens}(t) = \min\left(b,\ \text{tokens}(t^{-}) + r \cdot \Delta t\right)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
r	Tasa de relleno (ritmo sostenido permitido).	100 por minuto.
b	Tamaño del cubo (ráfaga máxima tolerada).	150
Δt	Tiempo transcurrido desde la última petición.	0,5 segundos.
$\text{tokens}(t)$	Crédito disponible ahora mismo.	Se gasta una unidad por petición.

En palabras: el cubo absorbe picos breves (hasta b) pero impone un ritmo medio (r) a largo plazo. Es la forma estándar de implementar un presupuesto por minuto que no castiga una ráfaga legítima ni deja que el sistema se desboque, y traduce directamente los límites TPM/RPM del proveedor a una política propia controlable.

Qué indican los proveedores y servicios

Los proveedores no dicen todos lo mismo, pero sus recomendaciones convergen en algo: **medir tokens, prefijos, latencia, cuota y modalidad de trabajo**.

PROVEEDOR O SERVICIO	QUÉ OFRECE O RECOMIENDA	USO EN ROUTING Y MÉTRICA CLAVE
OpenAI	Latency optimization, prompt caching automático, Batch API, Flex Processing y modelos pequeños para alto volumen.	Interactivo con streaming y caché; batch/flex para evals o enriquecimiento. Mira TTFT, tokens cacheados, output tokens, coste por aceptada y finalización batch.
Anthropic	Prompt caching automático o explícito, TTL de 5 minutos o 1 hora, cache diagnostics para misses.	Separar prefijo estable, tools e historial. Mira `cache_read_input_tokens`, `cache_creation_input_tokens`, TTFT y divergencia de prefijo.
OpenRouter	Routing por precio, throughput o latencia; percentiles recientes; `max_price`; filtros por parámetros; sticky routing y response caching.	Elegir proveedor por p90/p99, precio máximo o soporte de JSON/tools. Mira p50/p90/p99, throughput, proveedor elegido, cache hit y fallbacks.
LiteLLM	Load balancing entre despliegues, cooldowns, fallbacks, timeouts, retries, estrategias por peso, rate-limit aware, latencia o coste.	Gateway interno para unificar proveedores manteniendo política propia encima. Mira RPM/TPM por despliegue, cooldowns, retries y latencia añadida.
Runtime propio	vLLM, SGLang, TGI u otro serving controlado por el equipo.	Ruta local para datos sensibles, coste predecible o baja latencia si tienes capacidad. Mira TTFT, TPOT, memoria GPU, cola, KV cache y goodput.

OpenAI documenta que prompt caching funciona automáticamente en modelos recientes, que los hits dependen de prefijos exactos y que colocar contenido estático al principio ayuda a reducir latencia y coste.[21] También documenta Batch API para trabajos asíncronos con menor coste, límites separados y ventana de finalización de 24 horas; eso lo convierte en una ruta distinta, no en una optimización invisible.[22] Flex processing baja coste a cambio de más lentitud y posible falta temporal de recursos, así que debe vivir en rutas no urgentes y con timeouts acordes.[23]

Anthropic documenta prompt caching con caché automático o breakpoints explícitos, lectura de caché a menor coste y buenas prácticas como cachear contenido estable al principio del prompt.[24] También ofrece cache diagnostics para diagnosticar misses por divergencias de modelo, system prompt, tools o historial.[25] Esto es oro para ingeniería: si la métrica de caché cae, no quieres discutir; quieres saber qué byte cambió.

OpenRouter documenta routing por precio, throughput o latencia, preferencias por percentiles y filtros como `max_price` o soporte de parámetros.[26] También documenta prompt caching con sticky routing para mantener caliente el mismo proveedor cuando la caché aporta ahorro.[27] En su guía de latencia describe el overhead del gateway, cache warming, checks de saldo y fallback como factores operativos.[28]

LiteLLM documenta load balancing, cooldowns, fallbacks, timeouts, retries y estrategias como weighted pick, rate-limit aware, latency-based, least-busy y lowest-cost routing; además advierte que estrategias con tracking de uso pueden añadir latencia por Redis.[29] Esa advertencia es importante: una optimización puede introducir su propio coste.

Orden práctico para optimizar sin romper calidad

Si yo tuviera que optimizar un router de producción, no empezaría tocando pesos a ciegas. Seguiría este orden:

PASO	QUÉ HACES	POR QUÉ
1. Baseline	Mides p50/p95/p99, TTFT, TPOT, tokens, coste, contrato y aceptación por ruta.	Sin línea base no sabes si mejoras.
2. Separar clases	Distingues interactivo, batch, largo, RAG, tools y revisión.	Cada clase optimiza otra cosa.
3. Reducir salida	Ajustas contrato, longitud, schema y `max_output_tokens`.	Suele tocar latencia y coste de forma directa.
4. Estabilizar prefijos	Mueves instrucciones, ejemplos y tools estables al principio.	Mejora cache hit y reduce prefill repetido.
5. Podar entrada	Limpias HTML, deduplicas chunks, mejoras RAG y compaction.	Evita pagar contexto que no aporta.
6. Elegir modelo suficiente	Cambias a modelo menor solo si pasa evals.	Latencia y coste bajan sin convertirlo en lotería.
7. Usar rutas asíncronas	Batch o flex para lo que no exige respuesta inmediata.	Libera cuota interactiva y baja coste.
8. Optimizar proveedor	Ordenas por p90/p99, throughput, precio máximo y soporte de contrato.	Aprovechas señales vivas del mercado.
9. Probar política	Shadow routing, golden traces, canary y rollback.	Evita que una mejora media rompa casos críticos.

El criterio final no es “ha bajado la factura”. Es:

$$\Delta C_{util} < 0 \quad \wedge \quad \Delta L_{p95} \leq 0 \quad \wedge \quad \Delta Q \geq -\epsilon \quad \wedge \quad \Delta F_{contract} \leq \tau$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
ΔC_{util}	Cambio en coste por run aceptada.	Queremos que baje.
ΔL_{p95}	Cambio de latencia p95.	No debería empeorar en rutas interactivas.
ΔQ	Cambio de calidad evaluada.	Permitimos caída mínima ϵ si el producto lo acepta.
$\Delta F_{contract}$	Cambio en fallos de contrato.	No puede superar el umbral τ.
ϵ, τ	Tolerancias explícitas.	`quality_drop <= 0.01`, `contract_fail_delta <= 0.002`.

Esa fórmula evita una trampa muy común: declarar victoria porque el coste baja mientras la calidad cae o el contrato falla más.

Patrones que ayudan a no romper el sistema

PATRÓN	DECISIÓN CONCRETA
Cola por clase de tarea	Separar interactivo, batch, largo, RAG y revisión.
Presupuesto decreciente	Cada fase resta tiempo, coste y retries disponibles.
Fallback hacia menos alcance	Si no cabe la respuesta completa, entregar resumen breve o pasar a batch.
Circuit breaker por ruta	Si una ruta degrada, dejar de enviarle casi todo el tráfico.
Canary de rutas	Probar una ruta nueva con poco tráfico y comparar contra golden traces.
Shadow routing	Calcular qué habría elegido una política nueva sin ejecutar la ruta.
Holdout de evaluación	No optimizar el router solo con los mismos casos que usas para ajustarlo.
Idempotencia	Si reintentas, que no dupliques efectos externos.

El último punto importa mucho. Reintentar una lectura es una cosa. Reintentar una acción con efecto en sistemas externos es otra. Si una tool escribe, cobra, publica, borra o modifica estado, el retry debe pasar por idempotency keys, confirmación o estado explícito.

Cómo probar una política de routing

Una política de routing también se testea. No basta con mirar una demo.

PRUEBA	QUÉ COMPRUEBA	EJEMPLO
Unit test de restricciones	Una ruta incompatible nunca se elige.	Si pide citas, `local_fast` queda descartada.
Test de presupuesto	Coste, tokens y latencia bloquean rutas caras.	`frontier_review` no entra en `fast_triage`.
Test de salud	Cooldown y timeout ratio cambian decisión.	Si `rag_medium_json` degrada, pasa a batch o ruta breve.
Golden traces	La decisión esperada se mantiene en casos clave.	`support_summary` elige `rag_medium_json` con razón conocida.
Shadow routing	Política nueva calcula decisión sin ejecutar.	Comparas `policy@1.7` contra `policy@1.8`.
Canary	Política nueva recibe poco tráfico real.	5% de `support_summary` durante una hora.
Test de coste útil	No mejora solo precio por llamada.	Baja coste por token, pero sube fallo de contrato: no se acepta.

Los tests deberían producir un informe. Algo así:

```
policy: router@1.8.0
cases: 120
changed_decisions: 14
expected_changes: 12
unexpected_changes: 2
cost_per_accepted_delta: -8.4%
contract_failure_delta: +0.3%
latency_p95_delta_ms: -410
decision: canary, not full rollout
```

Ese informe se parece más a ingeniería de software que a “probemos el prompt nuevo”. Y esa es la idea: el router es código de producción.

En el día a día

El routing aparece la primera vez que tienes más de un modelo, o más de una ruta, y alguien pregunta cuál usar. La respuesta operable no es «el mejor», sino «el mejor para esta tarea y este presupuesto». Una clasificación trivial puede ir a un modelo barato y rápido; una consulta delicada, al modelo fuerte; y si el modelo fuerte se satura o devuelve un error, el

fallback decide si el sistema se degrada con elegancia (responde con el barato, o se abstiene) o se cae entero. Diseñar esa decisión por adelantado es la diferencia entre un mal día y una incidencia.

El presupuesto por tarea aparece cuando el coste se dispara y nadie sabe por qué. En un prototipo, da igual; en producción, una tarea que se permite reintentar sin límite, ampliar contexto sin tope o saltar siempre al modelo caro puede multiplicar la factura sin que la calidad mejore. Poner un presupuesto por tarea (tokens, llamadas, coste, latencia) convierte «se nos fue de las manos» en «esta tarea se detiene cuando llega a su límite», que es una frase que se puede defender ante quien paga.

Por qué debería importarte

Porque sin routing ni presupuesto un sistema de IA tiene dos finales malos, y los dos llegan pronto. O usa el modelo caro para todo y se vuelve insostenible en cuanto crece el tráfico, o usa el barato para todo y falla justo en las tareas que importan. El routing con fallback y presupuesto es lo que te permite la única respuesta sensata: gastar donde aporta y ahorrar donde no, con una red de seguridad cuando algo se rompe.

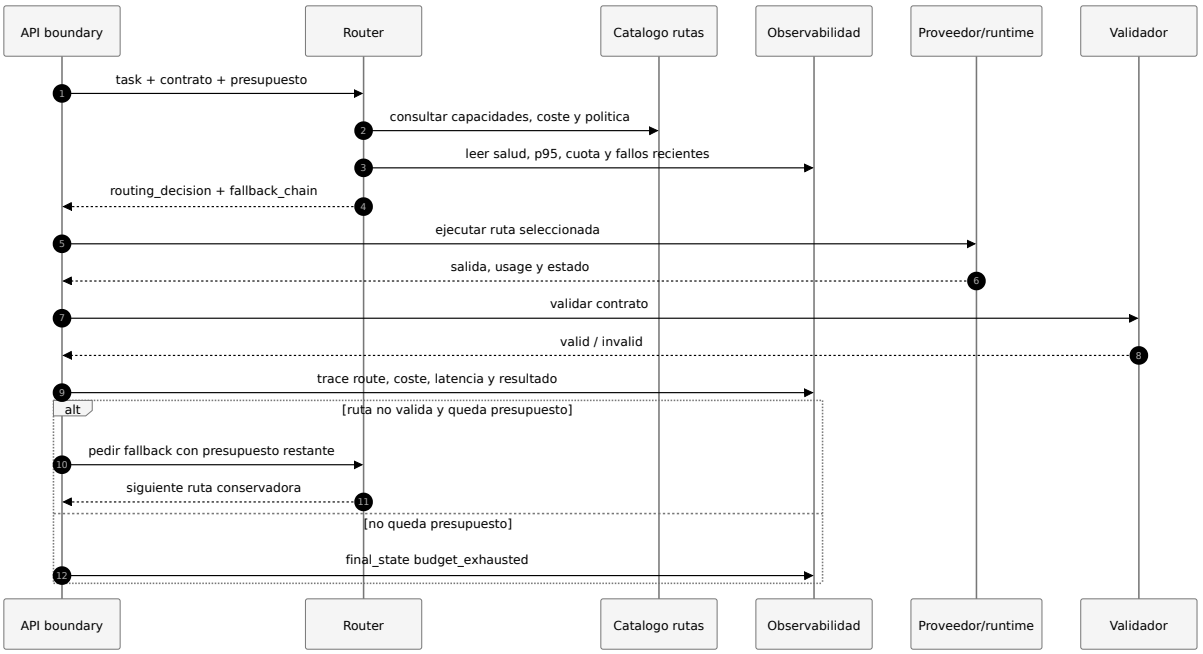
Y hay una consecuencia de oficio. Quien diseña el routing y el fallback puede prometer dos cosas que un sistema sin ellos no puede: un coste acotado y un comportamiento predecible cuando un proveedor falla. Esas dos promesas son las que distinguen un sistema que aguanta un pico de tráfico o una caída de un proveedor de uno que se desmorona con el primer imprevisto.

Dónde solía tropezar yo

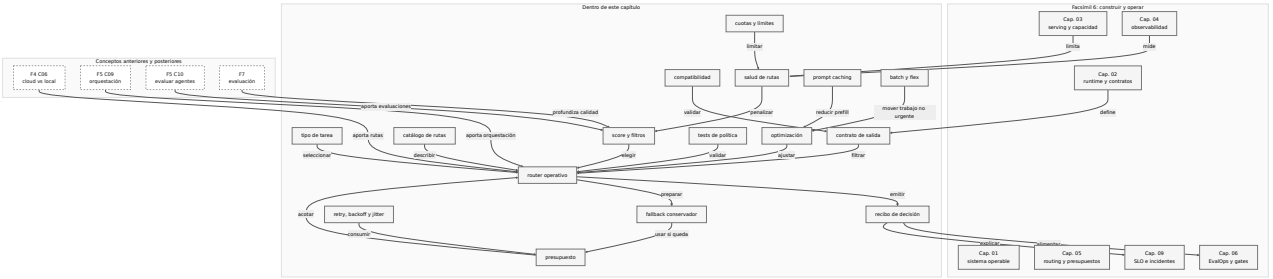
TROPIEZO	POR QUÉ ES UN PROBLEMA	ANTÍDOTO
Fallback como improvisación	Cambia comportamiento sin contrato y confunde al usuario.	Definir cadena de fallback por tarea y estado final si no cabe.
Reintentar todo	Aumenta coste y presión justo cuando una ruta está mal.	Retry solo para errores recuperables, con backoff, jitter y presupuesto.
Optimizar precio por llamada	Puede subir el coste útil si fallan más respuestas.	Medir coste por run aceptada y fallo de contrato por ruta.
No registrar alternativas descartadas	Luego nadie sabe por qué el router eligió una ruta.	Emitir `routing.decision` con `reason` y `alternatives`.
Tratar todas las tareas igual	Una tarea batch y una interactiva no tienen el mismo SLO.	Presupuesto por tarea y colas separadas.
Delegar toda la política en un agregador	Pierdes criterio de producto y trazabilidad interna.	Mantener router lógico propio aunque uses herramientas externas.
No probar la política	Cambia producción sin saber qué casos se mueven.	Unit tests, golden traces, shadow routing y canary.
Optimizar una métrica aislada	Baja coste o latencia, pero sube fallo de contrato o baja calidad.	Optimizar con coste útil, p95, calidad y contrato a la vez.
Romper el prefijo cacheable	Un timestamp o cambio de orden en tools puede destruir el cache hit.	Mantener prefijos estables y medir `cache_read_input_tokens`.

Cómo encaja todo

Primero, el ciclo operativo de una run con routing:



Y el mapa conceptual del capítulo:



Routing es donde se juntan tres mundos: producto, infraestructura y evaluación. Producto define qué necesita la tarea. Infraestructura dice qué capacidad existe. Evaluación dice qué ruta funciona de verdad.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Router	Componente que selecciona una ruta de ejecución.
Ruta	Combinación de modelo, proveedor, runtime, contrato, cola y política.
Catálogo de rutas	Registro versionado de rutas, capacidades, costes, límites, owner y fallback.
Fallback	Ruta alternativa usada cuando la principal no cumple o no conviene.
Presupuesto de tarea	Límite de coste, latencia, tokens, retries y acciones para una tarea concreta.
Cuota	Capacidad asignada por proveedor o runtime, medida en tokens, peticiones o concurrencia.
Rate limit	Límite operativo de uso por unidad de tiempo.
Matriz de compatibilidad	Tabla que compara qué rutas soportan contrato, tools, streaming, región y parámetros.
Retry	Nuevo intento controlado de una operación.
Backoff	Espera creciente entre retries.
Jitter	Aleatoriedad añadida al backoff para evitar reintentos sincronizados.
Circuit breaker	Mecanismo que deja de enviar tráfico a una ruta temporalmente degradada.
Cooldown	Periodo de espera antes de volver a probar una ruta.
Load shedding	Rechazar o retrasar trabajo para proteger el servicio.
Backpressure	Señal para que clientes o colas reduzcan entrada.
Coste útil	Coste por run aceptada, no solo coste por llamada.
Prompt caching	Reutilización de un prefijo estable del prompt para reducir latencia y coste de entrada.
Cache hit	Petición que encuentra en caché el prefijo esperado.
Batch	Procesamiento asíncrono de muchas peticiones que no necesitan respuesta inmediata.

TÉRMINO	DEFINICIÓN
Flex processing	Ruta de menor coste y menor prioridad, aceptando más espera o disponibilidad variable.
Sticky routing	Mantener una conversación o prefijo en el mismo proveedor para aumentar hits de caché.
Shadow routing	Evaluar qué ruta elegiría una política nueva sin ejecutarla.
Canary	Desplegar una ruta o política a una fracción pequeña del tráfico.
Golden trace	Traza representativa que sirve como referencia para comparar decisiones de routing.

Antes de pasar página

- [ ] ¿Puedes explicar por qué routing no es solo fallback?
- [ ] ¿Puedes definir un presupuesto de tarea con latencia, coste, tokens y retries?
- [ ] ¿Puedes decidir cuándo un retry tiene sentido y cuándo no?
- [ ] ¿Puedes explicar backoff y jitter sin fórmulas opacas?
- [ ] ¿Puedes distinguir coste por llamada de coste por run aceptada?
- [ ] ¿Puedes diseñar una cadena de fallback conservadora para una tarea concreta?
- [ ] ¿Puedes decir qué campos debería tener un evento `routing.decision`?
- [ ] ¿Puedes explicar por qué `run_id` no debería ser label de una métrica de routing?
- [ ] ¿Puedes conectar router con observabilidad, SLO y EvalOps?
- [ ] ¿Puedes justificar cuándo usar router propio, OpenRouter, LiteLLM o una mezcla?
- [ ] ¿Puedes diseñar un catálogo de rutas con capacidades, límites, owner y fallback?
- [ ] ¿Puedes calcular si una tarea cabe en una cuota TPM/RPM con margen?
- [ ] ¿Puedes explicar por qué una API compatible no garantiza comportamiento compatible?
- [ ] ¿Puedes proponer tests para una política nueva antes de desplegarla?
- [ ] ¿Puedes ordenar palancas de optimización sin tocar primero el modelo?
- [ ] ¿Puedes explicar cuándo usar prompt caching, batch, flex, streaming o response cache?
- [ ] ¿Puedes definir una condición de éxito que incluya coste, p95, calidad y contrato?

En resumen

IDEA	QUÉ DEBES LLEVARTE
Routing es una decisión operativa.	Combina tarea, contrato, presupuesto, salud, evaluación y trazabilidad.
El catálogo de rutas es el suelo del router.	Sin capacidades, owner, límites, región y fallback versionados, la decisión se vuelve opaca.
Fallback debe conservar contrato.	Una ruta alternativa no puede inventar permisos, formato o alcance.
El presupuesto vive durante la run.	Cada retry, tool, token y espera consume margen.
Las cuotas externas también son arquitectura.	RPM, TPM, concurrencia y región deben entrar antes de ejecutar.
Optimizar requiere elegir métrica objetivo.	Prompt caching, batch, flex, streaming, menor salida o modelo suficiente sirven para problemas distintos.
Reintentar mal empeora el sistema.	Timeout, backoff, jitter y circuit breaker protegen a usuarios y dependencias.
El coste útil manda más que el precio por token.	Una ruta barata puede salir cara si falla contrato o requiere revisión.
La decisión debe quedar explicada.	`routing.decision` permite aprender, depurar y mejorar la política.
Una política de routing se prueba como software.	Unit tests, shadow routing, canary y golden traces evitan cambios a ciegas.

Para saber más

- Amazon Web Services. (2026). *Timeouts, retries, and backoff with jitter*. AWS Builders Library. https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/
- Anthropic. (2026). *Cache diagnostics*. https://platform.claude.com/docs/en/build-with-claude/cache-diagnostics
- Anthropic. (2026). *Prompt caching*. https://platform.claude.com/docs/en/build-with-claude/prompt-caching
- Anthropic. (2026). *Rate limits*. https://platform.claude.com/docs/en/api/rate-limits
- Beyer, B., Jones, C., Petoff, J. y Murphy, N. R. (eds.). (2016). *Handling Overload*. En *Site Reliability Engineering*. https://sre.google/sre-book/handling-overload/

- Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. Communications of the ACM, 56(2), 74-80. https://doi.org/10.1145/2408776.2408794
- Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En B. Beyer, C. Jones, J. Petoff y N. R. Murphy (eds.), *Site Reliability Engineering*. https://sre.google/sre-book/monitoring-distributed-systems/
- LiteLLM. (2026). *Router - Load Balancing*. https://docs.litellm.ai/docs/routing
- Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$*. Operations Research, 9(3), 383-387. https://doi.org/10.1287/opre.9.3.383
- OpenAI. (2026). *Batch API*. https://developers.openai.com/api/docs/guides/batch
- OpenAI. (2026). *Flex processing*. https://developers.openai.com/api/docs/guides/flex-processing
- OpenAI. (2026). *Latency optimization*. https://developers.openai.com/api/docs/guides/latency-optimization
- OpenAI. (2026). *Prompt caching*. https://developers.openai.com/api/docs/guides/prompt-caching
- OpenAI. (2026). *Rate limits*. https://developers.openai.com/api/docs/guides/rate-limits
- OpenRouter. (2026). *Latency and Performance*. https://openrouter.ai/docs/guides/best-practices/latency-and-performance
- OpenRouter. (2026). *Prompt Caching*. https://openrouter.ai/docs/features/prompt-caching
- OpenRouter. (2026). *Provider routing*. https://openrouter.ai/docs/guides/routing/provider-selection
- OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. https://opentelemetry.io/docs/specs/semconv/gen-ai/
- Prometheus. (2026). *Metric and label naming*. https://prometheus.io/docs/practices/naming/

Notas

1. Beyer, B., Jones, C., Petoff, J. y Murphy, N. R. (eds.). (2016). *Handling Overload*. En *Site Reliability Engineering*. https://sre.google/sre-book/handling-overload/. Consultado el 27 de mayo de 2026. ↩

2. Amazon Web Services. (2026). *Timeouts, retries, and backoff with jitter*. AWS Builders Library. https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/. Consultado el 27 de mayo de 2026. ↩

3. OpenRouter. (2026). *Provider routing*. https://openrouter.ai/docs/guides/routing/provider-selection. Consultado el 27 de mayo de 2026. ↩

4. LiteLLM. (2026). *Router - Load Balancing*. https://docs.litellm.ai/docs/routing. Consultado el 27 de mayo de 2026. ↩

5. OpenAI. (2026). *Rate limits*. https://developers.openai.com/api/docs/guides/rate-limits. Consultado el 27 de mayo de 2026. ↩

6. Anthropic. (2026). *Rate limits*. https://platform.claude.com/docs/en/api/rate-limits. Consultado el 27 de mayo de 2026. ↩

7. Keeney, R. L. y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley. Formaliza la función de valor aditiva ponderada para decidir entre alternativas. ↩

8. OpenAI, 2026, *Rate limits*. ↩

9. Anthropic, 2026, *Rate limits*. ↩

.0. OpenRouter, 2026, *Provider routing*. ↩

.1. Dean, J. y Barroso, L. A. (2013). The Tail at Scale. *Communications of the ACM, 56*(2), 74-80. https://doi.org/10.1145/2408776.2408794. Consultado el 27 de mayo de 2026. ↩

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

.3. Prometheus. (2026). *Metric and label naming*. https://prometheus.io/docs/practices/naming/. Consultado el 27 de mayo de 2026. ↩

.4. OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. https://opentelemetry.io/docs/specs/semconv/gen-ai/. Consultado el 27 de mayo de 2026. ↩

.5. OpenAI. (2026). *Latency optimization*. https://developers.openai.com/api/docs/guides/latency-optimization. Consultado el 28 de mayo de 2026. ↩

.6. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H. y Stoica, I. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. *Proceedings of SOSP 2023*. https://doi.org/10.1145/3600006.3613165 Distingue las fases de prefill y decode que dominan la latencia de inferencia. ↩

.7. Trivedi, K. S. (2016). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications* (2.ª ed.). Wiley. Para componentes redundantes en paralelo, la fiabilidad del conjunto es el complemento del producto de las probabilidades de fallo individuales. ↩

.8. Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3), 235-256. https://doi.org/10.1023/A:1013689704352. Introduce UCB1 con garantías de arrepentimiento logarítmico. ↩

19. El número de ensayos de Bernoulli independientes hasta el primer éxito sigue una distribución geométrica de media $1/p$; véase Ross, S. M. (2014). *Introduction to Probability Models* (11.ª ed.). Academic Press. ↩

20. Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks* (5.ª ed.). Pearson. Describe el algoritmo del cubo de tokens (token bucket) para conformar tráfico permitiendo ráfagas hasta el tamaño del cubo. ↩

21. OpenAI. (2026). *Prompt caching*. https://developers.openai.com/api/docs/guides/prompt-caching. Consultado el 28 de mayo de 2026. ↩

22. OpenAI. (2026). *Batch API*. https://developers.openai.com/api/docs/guides/batch. Consultado el 28 de mayo de 2026. ↩

23. OpenAI. (2026). *Flex processing*. https://developers.openai.com/api/docs/guides/flex-processing. Consultado el 28 de mayo de 2026. ↩

24. Anthropic. (2026). *Prompt caching*. https://platform.claude.com/docs/en/build-with-claude/prompt-caching. Consultado el 28 de mayo de 2026. ↩

25. Anthropic. (2026). *Cache diagnostics*. https://platform.claude.com/docs/en/build-with-claude/cache-diagnostics. Consultado el 28 de mayo de 2026. ↩

26. OpenRouter, 2026, *Provider routing*. ↩

27. OpenRouter. (2026). *Prompt Caching*. https://openrouter.ai/docs/features/prompt-caching. Consultado el 28 de mayo de 2026. ↩

28. OpenRouter. (2026). *Latency and Performance*. https://openrouter.ai/docs/guides/best-practices/latency-and-performance. Consultado el 28 de mayo de 2026. ↩

29. LiteLLM, 2026, *Router - Load Balancing*. ↩

Cuadernos para practicar

Escanea el código con el móvil para abrir el cuaderno en Google Colab y ejecutarlo sin instalar nada.

Routing, fallback y presupuesto: el modelo adecuado para cada tarea

https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2006/02-routing-fallback-presupuesto.ipynb