El gateway: FastAPI en producción

Para escribir desde cero, con la documentación abierta, la API que está entre los clientes y el servidor de modelos, y para diagnosticar por qué va lenta, devuelve 503 o se traba. El gateway del C01 recibe la foto, la valida, la decodifica en CPU sin frenar a los demás requests, llama a Triton con un plazo, rechaza rápido lo que no puede atender a tiempo y deja una línea de log y sus métricas por cada request. Cada sección trae la sintaxis como plantilla con variables, las piezas, un ejemplo real del curso, una figura, un truco para recordarla, la trampa típica y un laboratorio opcional. En la versión web, al final hay un diagnosticador de síntomas, un revisor de código del gateway y un quiz de errores.

Cómo recordarla · Trampa típica · *variable* la cambias tú · Ejemplos en verde

EN LAS FIGURAS: CPU del gateway · espera: red, cola o turno · la GPU (Triton) · ok · rechazado a propósito: 4xx o 503 · falla: 500, 502, 504 o se corta · fuera del gateway

0 Números de bolsillo: códigos HTTP, timeouts, límites del gateway del C01 y Little's law

¿Qué códigos, tiempos y límites conviene saberse de memoria para leer una respuesta del gateway, o un incidente, en segundos?

SINTAXIS CON VARIABLES

curl -si -H "Content-Type: *tipo*" --data-binary @{*foto*} "*host*/v1/classify?top_k=*k*"

Manda una foto como lo hace el backend de publicación (el servidor del marketplace que llama al C01). Con `-i` ves la línea de estado (el código y su frase) y los headers: `X-Request-ID` viene en el 200, en los errores que decide la función y en el 422 de FastAPI, que pasa por su exception handler (un 500 por bug no lo trae), y `Retry-After` en los 503 de carga y de Triton. Si le hablas directo a uvicorn (el servidor que corre la app), la frase la pone él según tu Python: desde 3.13 dice `413 Content Too Large` y `422 Unprocessable Content`, los nombres del RFC 9110 (el documento que define qué significa cada código).

raise HTTPException(*codigo*, "*mensaje*", headers={"Retry-After": "*segundos*"})

Así responde el C01 cada error que decide: FastAPI lo convierte en `{"detail": "…"}` con ese código y esos headers. Un 4xx dice «el request está mal: no lo repitas igual» (el 429 es la excepción: «espera y reintenta»). Un 503 con `Retry-After` dice «estoy lleno o mi dependencia no está: vuelve en *segundos* s».

left = deadline - time.perf_counter()
resultado = await asyncio.wait_for(*llamada*, timeout=min(*tope*, left))

Deadline propagation (propagar la hora límite): el request tiene un presupuesto total (250 ms en el C01) y cada etapa usa lo que queda, nunca más que su propio tope (150 ms para Triton). Si ya no queda nada, el C01 responde 504 sin llamar a Triton.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
tipo	formato de la foto, en el header Content-Type	image/jpeg (también image/png e image/webp)
foto	archivo que mandas como body, en bytes crudos	foto.jpg
host	dónde escucha el gateway	localhost:8080 con compose; en EKS (el Kubernetes de AWS), el balanceador interno (ALB)
k	cuántas categorías pides	5 por defecto, de 1 a 20
codigo · *mensaje*	código HTTP y texto para el cliente	503 · gateway saturado: reintenta más tarde
segundos	cuánto esperar antes de reintentar	1 en el C01
llamada · *tope*	la espera lenta que acotas y el máximo propio de esa etapa	client.infer(...) · INFER_TIMEOUT_S = 0.15
L	requests en vuelo: aceptados y todavía sin respuesta, en cualquier etapa	≈ 3–5 por pod en el pico
λ	requests por segundo que llegan a un pod	500 req/s ÷ 5 pods = 100

`python3.13 -c 'from http import HTTPStatus; print(HTTPStatus(` *codigo* `).phrase)'`

La frase oficial de cada código en tu versión de Python; uvicorn la toma de ahí. Con 3.12 o anterior verás los nombres viejos: `Request Entity Too Large` y `Unprocessable Entity`.

VARIABLE	QUÉ ES	EJEMPLO
W	tiempo promedio que pasa cada request adentro (el promedio, no el p99 de los más lentos)	≈ 50 ms normal; 200 ms si todo va lento
$t_{\text{caller}} \dots t_{\text{cola}}$	timeout de cada nivel: quien llama, el gateway, la llamada a Triton y la cola de Triton	300 · 250 · 150 · 100 ms
t_0	el momento en que el request llegó a la función	time.perf_counter() al entrar a classify
r, n	reintentos por capa y cuántas capas reintentan por su cuenta	3 y 3 → hasta 64 intentos por request

FÓRMULAS

$$L = \lambda W \qquad 100\,\tfrac{\text{req}}{\text{s}} \times 0.05\ \text{s} = 5 \qquad 100\,\tfrac{\text{req}}{\text{s}} \times 0.2\ \text{s} = 20 \rightarrow \text{MAX_INFLIGHT} = 16$$

$$t_{\text{caller}} > t_{\text{deadline}} > t_{\text{Triton}} > t_{\text{cola}} \qquad 300 > 250 > 150 > 100\ \text{ms}$$

$$t_{\text{Triton}} = \min\left(150\ \text{ms}, t_0 + 250\ \text{ms} - \text{ahora}\right) \qquad \le 0 \Rightarrow 504\ \text{sin llamar}$$

$$\text{RAM} \approx \underbrace{80}_{\text{Python}} + \underbrace{2 \times 320}_{\text{decodes de 40 MP}} + \underbrace{16 \times 10}_{\text{bodies en vuelo}} = 880\ \text{MB} < 1\ \text{GiB} \approx 1{,}074\ \text{MB}$$

$$\text{intentos} = (1 + r)^n \qquad r = 3,\ n = 3 \Rightarrow 4^3 = 64$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
200 OK	Respuesta con las k categorías, su score, model_version y latency_ms.	header X-Request-ID para reclamar un caso
400 Bad Request	El request está roto: body vacío, bytes que no son una imagen, o una relación de aspecto mayor que 1:8.	no se pudo decodificar la imagen
413 Content Too Large	Demasiado grande: más de 10 MB (por Content-Length o contando los bytes) o más de 40 MP (40 millones de píxeles, leídos del header de la imagen).	la imagen supera 10 MB
415 Unsupported Media Type	Formato no aceptado: el Content-Type no es image/jpeg, image/png ni image/webp.	un GIF, o application/octet-stream
422 Unprocessable Content	Lo genera FastAPI cuando un parámetro no pasa la validación, antes de llamar a tu función. El body trae la lista de errores.	top_k=0 o top_k=500
499 (el cliente se fue)	No está en el RFC: es la convención de nginx para «el cliente cerró la conexión antes de la respuesta». El C01 lo anota cuando el cliente corta a mitad del upload; nadie recibe esa respuesta, y no cuenta como error del servidor.	el cliente cerró la conexión antes de terminar el upload
429 Too Many Requests	Límite por cliente (rate limit, del RFC 6585): «mandaste demasiados en este rato». Puede traer Retry-After.	el main.py del C01 no lo emite: la cuota por caller va en el borde (una regla del WAF, el firewall de aplicaciones de AWS) o en un contador por identidad. Ojo: el WAF bloquea con 403 salvo que le configures una respuesta 429
500 Internal Server Error	Algo que nadie atrapó: un bug. Lo responde la capa de Starlette (la base de FastAPI) que atrapa cualquier excepción suelta. El C01 también lo usa a propósito: si Triton devuelve logits con NaN, responde un 500 propio.	el C01 arranca su contador con status = 500, por si algo explota

PIEZA	QUÉ HACE	EJEMPLO
502 Bad Gateway	Quien está delante recibió una respuesta inválida, o una conexión cerrada, del que está detrás.	`el ALB, si uvicorn cierra una conexión ociosa (keep-alive) que el ALB iba a reusar: por eso --timeout-keep-alive 75 > 60`
503 + Retry-After: 1 (carga)	Load shedding (rechazar al entrar en vez de encolar): ya hay 16 requests en vuelo, o en 50 ms no se liberó ningún turno de CPU (slot) para decodificar.	`gateway saturado: reintenta más tarde`
503 + Retry-After: 1 (Triton)	Falló la dependencia: Triton no responde (error de conexión) o rechazó la inferencia.	`Triton no disponible`
503 de otros	/readyz sin modelo (lo lee el kubelet, el agente de Kubernetes que corre los probes, los chequeos de salud; va sin Retry-After); el ALB cuando no le queda ningún pod listo; uvicorn con --limit-concurrency (texto plano: Service Unavailable). Ninguno pasa por tu manejo de errores.	`el modelo no está listo en Triton`
504 Gateway Timeout	Se acabó el tiempo: Triton no respondió en min(150 ms, lo que queda), o los 250 ms se gastaron antes de llamarlo.	`timeout de inferencia`
Retry-After	Header que dice en cuántos segundos reintentar (o desde qué fecha).	`Retry-After: 1`
X-Request-ID	Id del request: el C01 usa el que trae el cliente o un proxy (un intermediario), y si no viene, inventa uno. Une la respuesta con su línea de log.	`abc123, o uuid4().hex`

LÍMITE	DEFAULT	EN EL C01	POR QUÉ
Requests en vuelo por proceso	sin límite (--limit-concurrency de uvicorn no tiene default)	MAX_INFLIGHT = 16 → `503` + Retry-After: 1	Little: ≈ 100 req/s × 0.2 s ≈ 20, recortado por RAM
Hilos del threadpool	40 (el de AnyIO: rutas y dependencias def, y run_in_threadpool)	40, pero solo 2 decodes a la vez (un semáforo: un contador de turnos)	un decode de 40 MP ocupa ≈ 320 MB
Espera por un slot de CPU	—	SLOT_WAIT_S = 50 ms → `503` + Retry-After: 1	esperar más ya rompe el SLO
Tamaño del body	sin límite propio (FastAPI, uvicorn y Starlette no traen un tope para el body completo: lo pones tú, como _read_body_limited)	10 MB → `413`, por Content-Length y contando bytes	la app móvil manda ≈ 150–250 KB
Píxeles	Pillow avisa sobre 89,478,485 px y falla sobre el doble	40 MP → `413` (guardia propia); Pillow a 40 MP y 80 MP	un PNG de 50 KB puede declarar 30,000 × 30,000 px
Relación de aspecto	—	hasta 1:8 → si no, `400`	1 × 4,000 px pasa el límite de píxeles y ocuparía ≈ 1 GB al reescalar
top_k	—	5; de 1 a 20 → si no, `422`	nadie necesita 500 sugerencias
Timeout hacia Triton	tritonclient aio: conn_timeout = 60 s, un mismo valor para todas las llamadas	asyncio.wait_for con min(150 ms, lo que queda); conn_timeout = 1 s de respaldo	el timeout de infer() son µs de la cola de Triton, no del cliente
Presupuesto del request	ninguno	REQUEST_DEADLINE_S = 250 ms → `504`	50 ms antes que los 300 ms del caller
Conexiones a Triton	conn_limit = 100	64	acota lo que va en vuelo hacia Triton
Readiness hacia Triton	—	is_model_ready con tope de 0.5 s	el probe del kubelet corta a los 2 s
Keep-alive de uvicorn	5 s	75 s (más que los 60 s de idle del ALB)	si uvicorn cierra primero, el ALB ve un 502

LÍMITE	DEFAULT	EN EL C01	POR QUÉ
Apagado de uvicorn	--timeout-graceful-shutdown: sin límite	20 s; preStop de 15 s; plazo del pod de 45 s	45 − 15 = 30 s para la app: 20 de drenado + 10 de margen
Procesos por contenedor	--workers 1 (o $WEB_CONCURRENCY)	--workers 1, escrito explícito	se escala con pods, no con procesos

Versiones de hoy contra las del C01: FastAPI 0.160.0 (0.141.1), Starlette 1.9.0 (1.7.0), Pydantic 2.14.1 (2.13.5), Pillow 13.0.0 (12.3.0), aiohttp 3.14.5 (3.14.3), tritonclient 2.73.0 (2.72.0), uvloop 0.23.0 (0.22.1), httptools 0.9.0 (0.8.0); uvicorn 0.54.0, AnyIO 4.15.1 y prometheus-client 0.26.0 siguen iguales. En el medio, FastAPI sumó soporte nativo de OpenTelemetry, el estándar de trazas y métricas (0.142; desde 0.143 el exportador ya no se configura solo) y dejó de validar los defaults que el cliente omite (0.148). Ningún cambio mueve un número de esta tabla.

Los límites del gateway: lo que trae cada librería por defecto contra lo que fija el C01. Verificado contra el código y la documentación de las versiones del C01 (FastAPI 0.141.1, Starlette 1.7.0, uvicorn 0.54.0, AnyIO 4.15.1, Pillow 12.3.0, tritonclient 2.72.0, Python 3.13), en octubre de 2026.

EJEMPLO REAL · LA CONFIGURACIÓN DEL C01: TODOS LOS NÚMEROS DE BOLSILLO EN UN SOLO LUGAR

PYTHON `reference/c01/gateway/app/main.py (líneas 41–60, extracto; # … marca líneas omitidas)`

```python
INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.15"))  # 150 ms para Triton dentro del p99 ≤ 200 ms
# …
# Decodes simultáneos por proceso: acota CPU y RAM. Pico medido de UN PNG de 40 MP: ≈ 320 MB
# (Pillow guarda RGB en 4 bytes/px = 160 MB, más las copias de exif_transpose y convert).
PREPROCESS_CONCURRENCY = int(os.getenv("PREPROCESS_CONCURRENCY", "2"))
# Load shedding: requests aceptados a la vez por proceso; el resto → 503 inmediato con Retry-After.
# ≈ lo que un pod completa en ≈ 200 ms (≈ 100 req/s × 0.2 s ≈ 20) y acotado por RAM: 16 bodies de
# hasta 10 MB = 160 MB, que caben en 1Gi junto a los decodes. Al pico hay ≈ 3–5 en vuelo por pod.
MAX_INFLIGHT = int(os.getenv("MAX_INFLIGHT", "16"))
SLOT_WAIT_S = 0.05  # espera máxima por un slot de CPU: si hay que esperar más, ya no cumple el SLO → 503
# Presupuesto total server-side (< 300 ms del caller). Cada etapa usa lo que queda: deadline propagation
REQUEST_DEADLINE_S = float(os.getenv("REQUEST_DEADLINE_S", "0.25"))
MAX_BYTES = 10 * 1024 * 1024  # 10 MB: tope de subida web (el app móvil manda ~150–250 KB)
MAX_PIXELS = 40_000_000  # 40 MP: por encima es casi seguro un error o un ataque
ALLOWED_TYPES = {"image/jpeg", "image/png", "image/webp"}  # los únicos formatos que decodificamos
# …
MAX_TOP_K = 20  # top_k acotado: nadie necesita 500 sugerencias y el log crecería sin control
# Segunda red de Pillow: avisa > 40 MP y falla > 80 MP aunque alguien se salte la guardia de decode()
Image.MAX_IMAGE_PIXELS = MAX_PIXELS
```

- `INFER_TIMEOUT_S` (150 ms) y `REQUEST_DEADLINE_S` (250 ms) son la mitad de la cadena de timeouts. Las otras dos puntas: los 300 ms del caller y los 100 ms que Triton deja esperar en su cola (en su `config.pbtxt`).
- `MAX_INFLIGHT = 16` sale de Little: un pod atiende ≈ 100 req/s; si cada request tardara 200 ms, habría ≈ 20 adentro. Se recorta a 16 por RAM: 16 bodies de hasta 10 MB son 160 MB.
- `PREPROCESS_CONCURRENCY = 2`: solo 2 decodes a la vez por proceso, aunque el threadpool (el grupo de hilos de reserva) tenga 40. Un PNG de 40 MP ocupa ≈ 320 MB al decodificarse, y el pod tiene 1 GiB.
- `SLOT_WAIT_S = 0.05` no se configura por variable de entorno: si un request espera más de 50 ms por un slot de CPU, ya no llega al SLO (la meta: p99 ≤ 200 ms, el tiempo bajo el que quedan 99 de cada 100 requests) y recibe 503.
- Lo que viene de `os.getenv` se cambia sin reconstruir la imagen: `k8s/gateway.yaml` fija `INFER_TIMEOUT_S`, `PREPROCESS_CONCURRENCY` y `MAX_INFLIGHT`. `Image.MAX_IMAGE_PIXELS` deja a Pillow como segunda red: avisa sobre 40 MP y falla sobre 80 MP.

CÓMO RECORDARLA

4xx: el cliente se equivocó, no lo repitas igual (salvo 429: espera el Retry-After). 503: vuelve en un rato (Retry-After). 504: se acabó el tiempo. 500: bug tuyo. Los números del C01: **16** en vuelo, **2** decodes, **10 MB**, **40 MP**, y timeouts **300 → 250 → 150 → 100 ms**.

TRAMPA

Leer todo 503 como «el gateway se cayó». En el C01 hay al menos cinco: saturado (16 en vuelo o sin slot de CPU), Triton caído, Triton rechazando, `/readyz` sin modelo y el del ALB cuando ya no queda ningún pod listo, que ni pasa por el gateway. El `detail` y el `Retry-After` dicen cuál es, y el ALB cuenta aparte los 503 que genera él mismo (`HTTPCode_ELB_5XX_Count`). La otra mitad de la trampa: responder 500 a un timeout del upstream (el servicio de atrás, aquí Triton); para eso existe el 504.

En el curso: M3 · Little's law: la cuenta que ordena todo · M3 · Presupuesto de latencia end-to-end · M3 · Timeouts, reintentos y hedging · C01 · Serving · C01 · Estimaciones back-of-the-envelope · Hoja · Formulario de estimaciones

La cadena: el viaje de un request dentro del gateway

¿Qué le pasa a un request, en orden, desde que uvicorn recibe los bytes hasta que sale la respuesta y queda su línea de log?

SINTAXIS CON VARIABLES

`uvicorn modulo:app --host 0.0.0.0 --port puerto --workers 1 --no-access-log`

uvicorn es el servidor ASGI (ASGI: la interfaz estándar entre un servidor y una app async de Python). Abre el puerto, lee el HTTP con httptools (una librería en C que lo interpreta rápido) y le pasa cada request a tu app dentro del event loop (el único hilo que atiende a todos los requests `async` por turnos). Con uvloop instalado, como en el C01, usa ese event loop, más rápido.

`@app.post("ruta")`
`async def funcion(request: Request, param: int = Query(default, ge=min, le=max)):`

FastAPI elige la función por método y ruta, y valida los parámetros **antes** de llamarla: un `top_k` fuera de rango recibe 422 sin que corra una sola línea tuya. Con `request: Request` lees headers y body a mano, como el C01.

`resultado = await run_in_threadpool(funcion_cpu, datos)`

El paso de CPU (decode, resize, normalizar) va a un hilo del threadpool, el grupo de hilos de reserva. Mientras ese hilo trabaja, el event loop sigue atendiendo a los demás requests.

```
try:
    ...                                  # admisión, validación, CPU, Triton
finally:
    REQUESTS.labels(status=str(status), source=source, model_version=MODEL_VERSION).inc()
    log.info(json.dumps(record, ensure_ascii=False))
```

El último eslabón: en el `finally` se cuenta la métrica y se escribe una línea JSON pase lo que pase, también en errores. Ojo: solo ve los requests que llegaron a tu función. El 422 no llega: el C01 lo anota aparte, con un exception handler.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modulo : *app*	archivo (como módulo de Python) y variable que tiene la app	`app.main:app`
puerto	puerto donde escucha	`8080`
ruta · *funcion*	path del endpoint y la función que lo atiende	`/v1/classify · classify`
param · *default* · *min* · *max*	parámetro de query, su valor por defecto y su rango	`top_k · 5 · 1 · 20`
funcion_cpu · *datos*	la función que bloquea y sus argumentos	`_decode_and_hash · los bytes de la foto`
resultado	lo que devuelve el hilo	`batch, digest`
$t_{\text{ALB}} \dots t_{\text{Python}}$	lo que tarda cada tramo, según el presupuesto del C01	`ver la figura`
t_{slot}	lo que espera un request por uno de los 2 turnos de CPU (slots) para decodificar	`≈ 0 normal; tope de 50 ms, después 503`
t_{red}, t_{cola}, t_{GPU}	las tres partes de la llamada a Triton: ida y vuelta, cola de batching y cómputo	`≈ 2–4 · 0–2 · 3–6 ms`
p50, p99	el tiempo bajo el que quedan la mitad, y 99 de cada 100 requests	`≈ 20–45 ms · ≈ 150 ms (estimación pesimista)`
SLO	la meta del servicio: p99 ≤ 200 ms, medido en el servidor	`alerta si se pasa 10 minutos seguidos`

FÓRMULAS

$$t_{\text{total}} \approx t_{\text{ALB}} + t_{\text{body}} + t_{\text{slot}} + t_{\text{decode}} + t_{\text{Triton}} + t_{\text{post}} + t_{\text{Python}}$$

$$t_{\text{Triton}} = t_{\text{red}} + t_{\text{cola}} + t_{\text{GPU}} \approx (2\text{–}4) + (0\text{–}2) + (3\text{–}6)\ \text{ms en el p50}$$

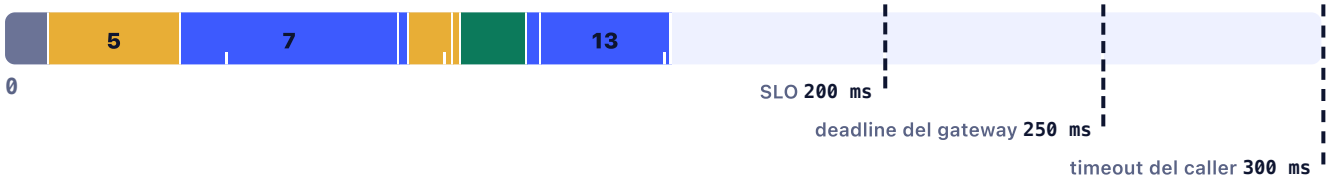
$$\text{p50} \approx 20\text{–}45\ \text{ms} \qquad \text{p99 pesimista} \approx 150\ \text{ms} < \text{SLO} = 200\ \text{ms}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
ALB	El balanceador de AWS: descifra la conexión (TLS), verifica el JWT (un token firmado que prueba quién llama) y elige un pod listo.	`solo deja pasar /v1/classify`
uvicorn	Servidor ASGI: acepta la conexión, lee el HTTP y llama a la app. Un proceso por contenedor.	`--workers 1 --no-access-log`
event loop (uvloop)	El único hilo que atiende a todos los requests async, por turnos; uvloop es una versión más rápida.	`uvicorn lo usa solo si uvloop está instalado`
ServerErrorMiddleware	La capa de más afuera (un middleware: una capa que envuelve a toda la app). Atrapa cualquier excepción suelta y responde 500.	`el C01 no agrega middlewares propios`

PIEZA	QUÉ HACE	EJEMPLO
ExceptionMiddleware	Convierte un HTTPException en una respuesta JSON con su código y sus headers.	`{"detail": "gateway saturado: reintenta más tarde"}`
Router y validación	Elige la función por método y path, y valida los parámetros con Pydantic (la librería que valida datos según tus tipos). Si fallan: 422, y tu función no corre.	`top_k=0 → 422`
Admisión	Control de admisión: si ya hay 16 requests en vuelo, 503 inmediato, sin leer el body.	`state.inflight >= MAX_INFLIGHT`
Validación barata	Content-Type (415) y Content-Length (413) se miran en los headers, sin leer un byte del body.	`image/jpeg, image/png, image/webp`
Lectura del body	request.stream(): el body llega por partes, y se corta apenas pasa de 10 MB.	`_read_body_limited`
Presupuesto	Antes de cada etapa cara se mira cuánto queda de los 250 ms; si no queda nada, 504.	`_budget_left(deadline)`
Slot de CPU + threadpool	Un semáforo (un contador de turnos) deja 2 decodes a la vez; el decode corre en un hilo aparte.	`await run_in_threadpool(_decode_and_hash, data)`
Llamada a Triton	Un cliente async por proceso, tensores en binario y timeout de min(150 ms, lo que queda).	`_infer → 504 o 503`
Postproceso y respuesta	Softmax, top-k, nombres de categoría, model_version y latency_ms, con el header X-Request-ID.	`{"model_version": …, "predictions": […]}`
Métricas y log	En el finally: contador por status, histograma de latencia y una línea JSON por request.	`classifier_requests_total{status="503"}`

Los p99 de los pasos, uno detrás del otro: ≈ **150 ms** · en el p50, 20–45 ms



A escala, de 0 a 300 ms; cada color es un paso de la lista, con su número. «Espera de un slot de CPU» (paso 6) no entra en la suma: es un tope, no una medida.

Paso a paso: barras a escala de 0 a 50 ms · p50 · p99

1 ALB + routing
borde (AWS)

p50 **1–3** p99 **5–10** ms

TLS ya terminado; conexiones abiertas que se reusan (keep-alive)

2 uvicorn + validación de top_k
gateway: FastAPI

µs microsegundos: no se ve a esta escala

SI NO PASA: 422

antes de tu función: lo anota un exception handler aparte

3 Admisión
gateway: classify

µs microsegundos: no se ve a esta escala

SI NO PASA: 503 + Retry-After: 1

16 en vuelo por proceso

4 Content-Type y Content-Length
gateway: _classify

µs microsegundos: no se ve a esta escala

SI NO PASA: 415 413

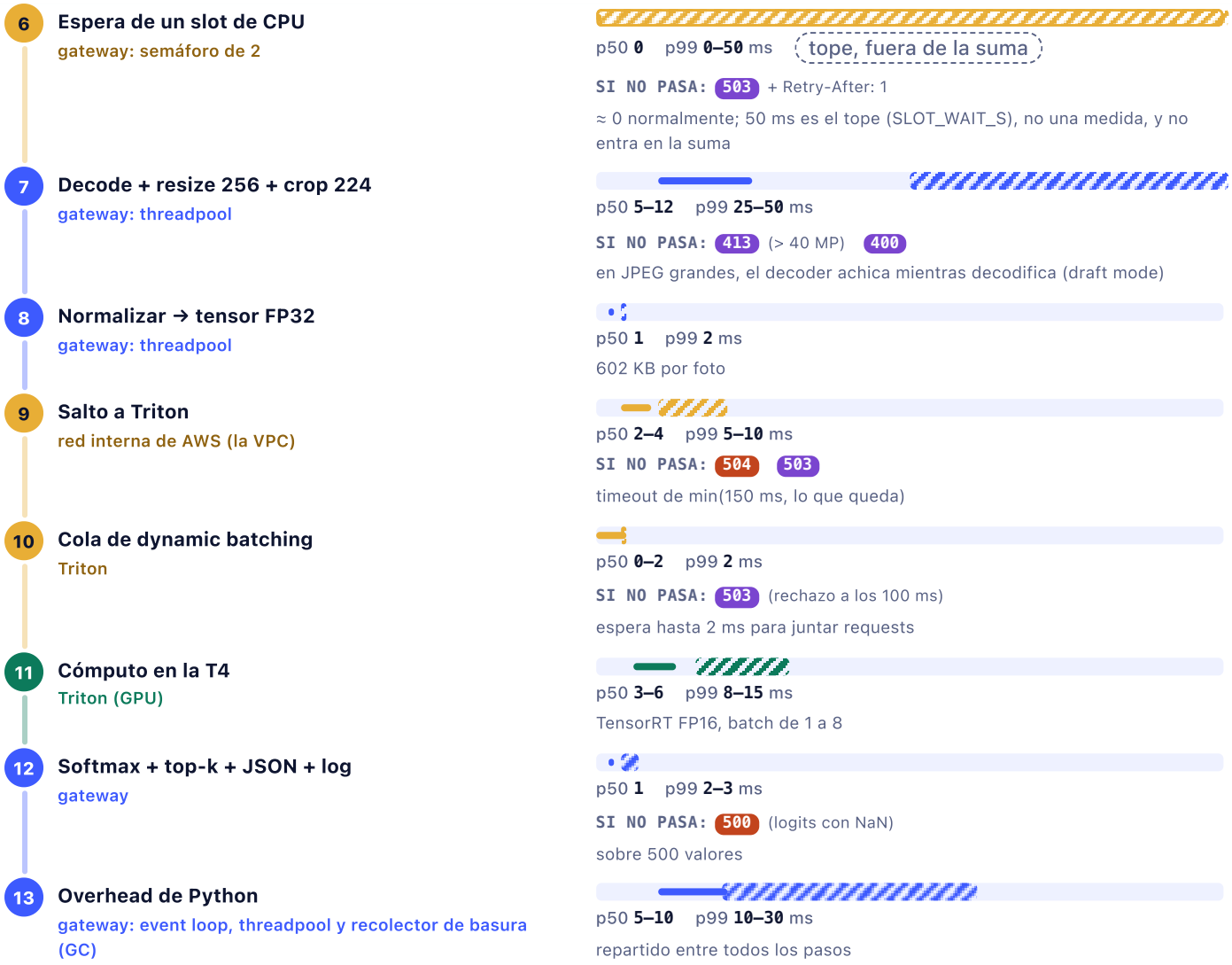
sin leer el body

5 Lectura del body
gateway: _read_body_limited

p50 **1–3** p99 **10–30** ms

SI NO PASA: 413 400

200 KB del celular contra hasta 10 MB de la web; si el cliente corta a la mitad, se anota como 499



Un request de /v1/classify por dentro, con el presupuesto del C01 (valores típicos a medir, no benchmarks). Las barras van a escala con el p99; las revisiones baratas cuestan microsegundos. Los p99 no se suman de verdad: su suma (≈ 150 ms) es una estimación pesimista que deja ≈ 50 ms de margen al SLO de 200 ms.

EJEMPLO REAL · CLASSIFY: ADMISIÓN, LA CADENA CON SU DEADLINE, Y EL FINALLY QUE SIEMPRE ANOTA

PYTHON reference/c01/gateway/app/main.py (líneas 206–249, extracto; # … marca líneas omitidas)

```python
@app.post("/v1/classify")
async def classify(request: Request, response: Response, top_k: int = Query(5, ge=1, le=MAX_TOP_K)):
    """Body = bytes crudos de la imagen (no multipart: menos parsing, sin archivos temporales)."""
    t0 = time.perf_counter()  # el reloj arranca con el request ya recibido (el SLO es server-side)
    # Si el cliente (o un proxy) ya trae un id lo respetamos: así se traza el request de punta a punta
    request_id = request.headers.get("x-request-id") or uuid.uuid4().hex
    # …
    status = 500  # si algo inesperado explota, se cuenta como 500
    try:
        state = request.app.state
        if state.inflight >= MAX_INFLIGHT:  # fallar rápido (µs) > encolar lo que va a vencer de todos modos
            raise HTTPException(503, "gateway saturado: reintenta más tarde", headers={"Retry-After": "1"})
        state.inflight += 1
        try:
            result = await _classify(request, top_k, record, deadline=t0 + REQUEST_DEADLINE_S)
        finally:
            state.inflight -= 1  # siempre, también si _classify lanza
        status = 200
        response.headers["X-Request-ID"] = request_id  # el cliente puede reportar un caso con este id
        result["latency_ms"] = round((time.perf_counter() - t0) * 1000, 1)
        return result
```

```
    except HTTPException as exc:
        status = exc.status_code
        record["error"] = exc.detail
        # Re-lanzamos con el X-Request-ID para que también los errores sean trazables
        headers = {**(exc.headers or {}), "X-Request-ID": request_id}
        raise HTTPException(exc.status_code, exc.detail, headers=headers) from None
    finally:
        elapsed = time.perf_counter() - t0
        # …
        REQUESTS.labels(status=str(status), source=source, model_version=MODEL_VERSION).inc()
        if not 400 <= status < 500:  # los 4xx son rechazos rápidos: incluirlos "mejoraría" el p99 falsamente
            LATENCY.labels(source=source).observe(elapsed)
        # …
        log.info(json.dumps(record, ensure_ascii=False))
```

- Antes de la primera línea de `classify` ya pasaron uvicorn, los middlewares y la validación de `top_k` (`Query(5, ge=1, le=MAX_TOP_K)`). Un 422 se responde ahí y no llega a este código: su línea y su conteo los escribe un exception handler aparte (`validation_error`).
- La admisión es la primera decisión de la función: con 16 en vuelo responde 503 en microsegundos, sin leer el body ni gastar CPU. El contador no necesita lock (un candado entre hilos): lo toca un solo hilo, el event loop.
- `t0 + REQUEST_DEADLINE_S` es el deadline (la hora límite) que viaja a `_classify` . Ahí se revisan los headers (415 y 413), se lee el body, se espera el slot de CPU, se decodifica en el threadpool y se llama a Triton. Antes del decode y antes de Triton se mira cuánto queda: si nada, 504; Triton recibe min(150 ms, lo que queda).
- Los errores se vuelven a lanzar con `X-Request-ID` : el cliente puede reclamar un 503 o un 504 con su id, y tú lo encuentras en el log.
- El `finally` corre siempre: cuenta el request por status, por quién llama (caller) y por modelo, mide la latencia (sin los 4xx, que «mejorarían» el p99 falsamente) y escribe una línea JSON sin la imagen ni datos personales.

CÓMO RECORDARLA

Lo barato primero, la CPU después, la GPU al final, y el log siempre. Cada eslabón puede cortar con su código (422, 503, 415, 413, 400 o 499, 504 y 500), y el `finally` anota todo lo que llegó a `classify` ; el 422 no llega, y lo anota su exception handler. Si el p99 sube, la cadena te dice dónde mirar: el body, el decode o Triton.

TRAMPA

Creer que tu métrica y tu log ven todos los requests. El 422 de `top_k` lo responde FastAPI antes de llamar a `classify` : no pasa por su `finally` . Por eso el C01 registra un exception handler (una función que FastAPI llama ante ese error) que escribe su línea y lo cuenta. Un 405 (un GET a `/v1/classify`) no pasa ni por ahí, y el C01 apagó el access log de uvicorn (la línea que uvicorn escribe por cada request: `--no-access-log`): ese solo lo ve el ALB, en sus 4xx.

En el curso: C01 · Arquitectura · C01 · Estimaciones back-of-the-envelope · M3 · Presupuesto de latencia end-to-end · C01 · Serving · C01 · Archivos de referencia · Hoja · Del dato a producción

1 Qué hace un gateway y por qué existe

¿Por qué hay un servicio tuyo entre los clientes y Triton, y qué trabajo le toca a cada capa?

SINTAXIS CON VARIABLES

```
POST /v1/classify?top_k=k
Content-Type: tipo

<bytes de la imagen>

→ 200 {"model_version": "version", "predictions":
[{"label": "categoria", "score": score}],
"latency_ms": ms}
```

El contrato (lo que tu API promete a quien la llama): los bytes crudos de la foto en el body, el top-k en la query y una respuesta con `model_version`. Un 200 nunca trae tensores ni logits (los puntajes crudos del modelo), ni los nombres del modelo o de los tensores en Triton.

```
VARIABLE = os.getenv("VARIABLE", "default")
```

Configuración por variables de entorno: la misma imagen corre en tu máquina, en staging (el ambiente de pruebas) y en producción, y Kubernetes le pone los valores. En el C01: `TRITON_URL`, `MODEL_VERSION`, `INFER_TIMEOUT_S`, `MAX_INFLIGHT`, entre otras.

```
app.state.triton =
triton_http.InferenceServerClient(url="host:8000",
conn_limit=n, conn_timeout=s)
```

Hacia atrás, el gateway habla KServe v2 (un protocolo estándar de servidores de modelos, el que habla Triton) con un solo cliente por proceso, creado en el lifespan (el código que corre al arrancar y al apagar el proceso). La URL va sin `http://`.

```
- path: /v1/classify
  pathType: Exact
  backend:
    service:
      name: servicio
```

Hacia adelante, el Ingress (la regla que le dice al ALB, el balanceador de AWS, qué rutas publicar) solo expone la API. `/metrics`, `/healthz`, `/readyz` y `/docs` existen en el pod, pero no salen del cluster.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
k · *tipo*	cuántas categorías pide el cliente y el formato de la foto	`5 · image/jpeg`
version · *categoria* · *score* · *ms*	lo que vuelve: versión del modelo, nombre de la categoría, su probabilidad y la latencia en el servidor	`2026-09-01-resnet50-fp16 · Celulares y telefonía > Smartphones · 0.83 · 41.2`
VARIABLE · *default*	nombre de la variable de entorno y el valor si no está	`INFER_TIMEOUT_S · 0.15`
host · *n* · *s*	dónde está Triton, conexiones máximas y timeout de respaldo	`triton.classifier.svc.cluster.local · 64 · 1.0`
servicio	el Service de Kubernetes del gateway	`gateway`
λ	requests por segundo en el pico de diseño	`500`
t_{CPU}	CPU por foto: decode, resize y normalizar	`≈ 6 ms con la mezcla de fotos del C01`
0.6×1 core	la meta del HPA (el autoscaler de pods de Kubernetes): agrega pods cuando el uso promedio pasa del 60 % de la CPU pedida	`0.6 cores por pod`
B	bytes del tensor que viaja a Triton	`602,112 (FP32: 3 × 224 × 224 × 4)`

FÓRMULAS

$$\text{cores de decode} = \lambda \cdot t_{\text{CPU}} = 500 \frac{\text{req}}{\text{s}} \times 6 \text{ ms} = 3$$

$$\text{pods del gateway} \approx \frac{\text{cores}}{0.6 \times 1 \text{ core por pod}} = \frac{3}{0.6} = 5 \qquad \text{(el HPA va de 3 a 12)}$$

$$\text{red gateway} \rightarrow \text{Triton} = \lambda \times B \times 8 = 500 \times 602{,}112 \text{ B} \times 8 \approx 2.4 \text{ Gb/s}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
TLS y autenticación	Cifrar la conexión y verificar quién llama, con un JWT de servicio (un token firmado) o mTLS (TLS con certificado en los dos lados).	`en el ALB, antes del gateway`
Identidad, no un header	x-client-source lo escribe el propio cliente: sirve para partir las métricas, nunca para cuotas ni permisos.	`source: mobile, web, internal u other`

PIEZA	QUÉ HACE	EJEMPLO
Cuota por caller	Limitar cuántos requests manda cada cliente, con 429 cuando se pasa.	**en el borde (una regla del WAF, el firewall de aplicaciones de AWS) o un contador por identidad; el main.py no la trae (el WAF responde 403 salvo que le configures 429)**
Validar barato	Rechazar lo inválido antes de gastar CPU: tipo, bytes, píxeles, top_k.	**415, 413, 400, 422**
Decode y preproceso	Convertir los bytes en el tensor que espera el modelo, con el mismo código del entrenamiento.	**preprocess.py → float32 [3, 224, 224]**
Separar CPU de GPU	El decode corre en nodos de CPU baratos que escalan solos; Triton solo recibe tensores.	**HPA por CPU al 60 % (gateway); KEDA, que escala según una métrica, por tiempo en cola (Triton)**
Admisión y backpressure	No aceptar más de lo que se puede terminar a tiempo, y avisar hacia atrás con 503 y Retry-After.	**MAX_INFLIGHT = 16 y 2 slots de CPU**
Timeouts y deadline	Cada etapa usa lo que queda del presupuesto: no se trabaja para un caller que ya se fue.	**250 ms en total, 150 ms hacia Triton**
Traducir errores y degradar	Convertir cada falla de atrás en un código útil y rápido, para que el cliente degrade con gracia: en el C01, la app muestra el selector manual de categorías y el vendedor publica igual.	**Triton lento → 504; caído o rechazando → 503 + Retry-After**
Postproceso	Logits → probabilidades → top-k → nombres. Cambia con el producto, no con el modelo.	**softmax y labels.json en el gateway**
Contrato y versión	Una API estable (/v1) aunque cambie el modelo; cada respuesta dice qué modelo contestó.	**model_version en el body**
Observabilidad	Una línea de log y métricas por request, con el tiempo de cada etapa.	**preprocess_ms e inference_ms por separado**
Salud	Liveness: ¿el proceso responde? Readiness: ¿puede atender? (aquí: ¿el modelo está listo en Triton?).	**/healthz y /readyz**
Lo que NO hace	Batching, colas por modelo y GPU: eso es de Triton. Y Triton nunca queda expuesto a los clientes.	**una NetworkPolicy (una regla de red de Kubernetes) solo deja entrar a Triton a los pods del gateway**

RESPONSABILIDAD	ALB (BORDE)	GATEWAY (FASTAPI, CPU)	TRITON (GPU)
TLS y autenticación	termina TLS y verifica el JWT	—	—
Cuota por caller	una regla del WAF (opcional)	contador por identidad (no está en la referencia)	—
Validar tipo, bytes y píxeles	—	`415` · `413` · `400`, antes de decodificar	revisa nombre, tipo y shape del tensor
Decode y preproceso	—	preprocess.py en el threadpool (hilos aparte)	—
Batching	—	una foto por request	dynamic batching: junta requests hasta 2 ms
Inferencia	—	—	engine TensorRT FP16
Postproceso	—	softmax, top-k y labels	devuelve logits
Admisión y backpressure	—	16 en vuelo y 2 slots de CPU → `503`	rechaza lo que espera 100 ms en cola
Timeouts	idle timeout de 60 s	deadline de 250 ms; 150 ms a Triton → `504`	100 ms de cola
Salud	health check a /readyz	/healthz y /readyz	/v2/health/live y /v2/health/ready
Métricas	5xx y latencia en CloudWatch (el monitoreo de AWS)	classifier_* en /metrics	nv_* en el puerto 8002

RESPONSABILIDAD	ALB (BORDE)	GATEWAY (FASTAPI, CPU)	TRITON (GPU)
Escala con	—	CPU: HPA al 60 %, de 3 a 12 pods	tiempo en cola: KEDA, de 2 a 4 pods

La línea del corte: el gateway sabe del producto (callers, formatos, top-k, errores); Triton, solo de tensores.

Quién hace qué en el camino de un request del C01. Una fila por responsabilidad; un guion quiere decir que esa capa no la toca.

EJEMPLO REAL · MAIN.PY SE PRESENTA SOLO: POR QUÉ EXISTE Y QUÉ ENDPOINTS TIENE

PYTHON `reference/c01/gateway/app/main.py (líneas 1–8 y 38–41; # … marca líneas omitidas)`

```python
"""Gateway HTTP del clasificador de categorías (caso c01).

Por qué existe un gateway en CPU delante de Triton: validar, decodificar y redimensionar imágenes es
trabajo de CPU; si lo hiciera Triton (python backend o DALI) competiría con la GPU cara. Así cada
capa escala por separado: gateway por CPU (HPA) y Triton por cola de GPU (KEDA).

Endpoints: POST /v1/classify?top_k=5 · GET /healthz (liveness) · GET /readyz (readiness) · GET /metrics
"""
# …
TRITON_URL = os.getenv("TRITON_URL", "localhost:8000")  # host:puerto HTTP de Triton, SIN "http://" (lo exige el cliente)
MODEL_NAME = os.getenv("MODEL_NAME", "classifier")  # nombre del modelo en el model repository de Triton
MODEL_VERSION = os.getenv("MODEL_VERSION", "dev")  # se devuelve y se loguea: cada predicción queda atada a su modelo
INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.15"))  # 150 ms para Triton dentro del p99 ≤ 200 ms
```

- La razón en una frase: decodificar y redimensionar es CPU, y la CPU del nodo con GPU es poca y cara. Separadas, cada capa escala con su propia señal: el gateway por CPU con el HPA, Triton por tiempo en cola con KEDA (el que escala según una métrica de Prometheus).
- Esos cuatro endpoints son todos los que escribe el C01 (FastAPI suma /docs, /redoc y /openapi.json). Solo `/v1/classify` sale del cluster; `/healthz` y `/readyz` son para los probes (los chequeos de salud) del kubelet, el agente de Kubernetes en cada nodo, y el ALB también mira `/readyz`; `/metrics` es para Prometheus (el que junta las métricas).
- `TRITON_URL` va sin `http://` porque así lo exige tritonclient; en EKS (el Kubernetes de AWS) apunta al Service de Triton (su dirección estable dentro del cluster) y en compose a `triton:8000`.
- `MODEL_VERSION` no la decide el gateway: viene de la release, vuelve en cada respuesta y queda en cada línea de log, para unir una predicción con su modelo.
- El precio de separar: un salto de red con 602 KB por foto (≈ 2.4 Gb/s en el pico, 3× el JPEG del celular), un componente más que operar y la obligación de rutear por zona (cada gateway le habla al Triton de su misma zona de AWS, porque el tráfico entre zonas se paga). Con poco tráfico, un solo proceso con el modelo adentro es la respuesta correcta: el MVP del C01, su primera versión.

CÓMO RECORDARLA

El gateway sabe del producto; Triton, solo de tensores. Lo de CPU va en CPU barata, lo de GPU en la GPU cara, y el cliente solo ve un contrato estable con `model_version`.

TRAMPA

Pensar el gateway como «un proxy que reenvía». Si solo reenvía, no protege nada: una ráfaga de fotos de 10 MB, un PNG de 50 KB que declara 30,000 × 30,000 px o un Triton lento tumban el servicio entero. Y la trampa inversa: exponer Triton a los clientes «para ahorrar un salto». Triton deja abiertas por defecto APIs de administración (el repositorio de modelos, los logs, las trazas) y no sabe de cuotas ni de identidades; vive detrás del gateway.

En el curso: M7 · Qué hace un model server (y qué no) · C01 · Arquitectura · C01 · Serving · M15 · Autenticación y autorización entre servicios · M15 · Rate limiting y protección contra abuso

2 FastAPI básico: app, rutas, modelos, response_model, códigos, dependencias y lifespan

¿Qué piezas de FastAPI vas a escribir en cualquier gateway, y qué hace cada una por ti?

SINTAXIS CON VARIABLES

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.recurso = crear(...)      # una vez por proceso, antes del tráfico
    yield                                # aquí la app atiende requests
    await app.state.recurso.cerrar()    # al apagar

app = FastAPI(title="nombre", lifespan=lifespan)
```

El lifespan (el código de arranque y apagado del proceso) reemplaza a `@app.on_event`, que está deprecado (marcado para desaparecer). Lo de antes del `yield` corre antes de aceptar requests; lo de después, cuando el servidor ya dejó de atenderlas. `app.state` guarda lo compartido: clientes, labels, un modelo.

```
class Salida(BaseModel):
    model_version: str
    predictions: list[Prediccion]

@app.post("/v1/ruta")
async def funcion(request: Request, top_k: Annotated[int, Query(ge=1, le=max)] = default) -> Salida:
    ...
```

Una ruta (path operation, en la doc de FastAPI) es una función con un decorador que dice el método y el path. Los tipos hacen el trabajo: `Query(ge=1, le=20)` valida la query y responde 422 si no cumple; el tipo de retorno (o `response_model=`) valida lo que devuelves, filtra lo que sobra y lo documenta en OpenAPI (la descripción de tu API, que FastAPI genera sola).

```
def get_recurso(request: Request) -> Tipo:
    return request.app.state.recurso

Alias = Annotated[Tipo, Depends(get_recurso)]

@app.post("/v1/ruta")
async def funcion(nombre_param: Alias): ...
```

Una dependencia (dependency) es una función que FastAPI llama por ti antes de la ruta, y cuyo resultado te pasa como parámetro: el cliente de Triton, la identidad ya verificada de quien llama, una sesión. Se llama una vez por request aunque varias partes la pidan; si la escribes con `def`, corre en el threadpool (el grupo de hilos de reserva).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
recurso · *crear* · *cerrar*	lo que creas una vez por proceso y cómo se cierra	`triton` · `InferenceServerClient` · `close`
nombre	título de la app en OpenAPI	`classifier-gateway`
Salida · *Prediccion*	modelos de Pydantic (clases que declaran campos con tipos) de la respuesta	`Salida con model_version y predictions`
funcion · *ruta*	la función que atiende y su path	`classify` · `classify`
max · *default*	rango y valor por defecto del parámetro	`20` · `5`
Tipo · *Alias* · *nombre_param*	lo que entrega la dependencia, un nombre corto para reusarla y el parámetro que la recibe	`InferenceServerClient` · `Triton` · `triton`
codigo · *mensaje*	código HTTP y texto para el cliente	`503` · `Triton no disponible`
Excepcion · *manejador*	el tipo de error que traduces y la función que responde	`ImageTooLarge` · `too_large`
c	el código que pusiste en la HTTPException	`413`

```
raise HTTPException(status_code=codigo,
detail="mensaje", headers={"Retry-After": "1"})

@app.exception_handler(Excepcion)
async def manejador(request: Request, exc:
Excepcion) -> JSONResponse: ...
```

Un error que decides se lanza como `HTTPException`: FastAPI responde `{"detail": …}` con ese código y esos headers. Para traducir una excepción tuya en todas las rutas a la vez, registras un exception handler (una función que FastAPI llama ante ese tipo de error). El C01 registra uno para el 422 de FastAPI: así también deja su línea de log y su conteo.

FÓRMULAS

$$\text{respuesta} = \text{JSON}\Big(\text{filtrar}_{\text{modelo de salida}}\big(\text{tu_función}(\text{validar}(\text{request}))\big)\Big)$$

$$\text{entrada inválida} \Rightarrow 422 \qquad \text{HTTPException}(c) \Rightarrow c \qquad \text{salida inválida o excepción suelta} \Rightarrow 500$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
FastAPI()	La app: junta rutas, validación, OpenAPI y el lifespan. Por debajo es una app de Starlette, el framework ASGI (la interfaz estándar entre un servidor y una app async) sobre el que está hecha.	`FastAPI(title="classifier-gateway", lifespan=lifespan)`
@app.get / @app.post	Decorador que registra una ruta: método HTTP + path → tu función.	`@app.post("/v1/classify")`
Parámetros de path y query	Lo que viene en la URL. FastAPI lo convierte al tipo que declaras y lo valida.	`top_k: int = Query(5, ge=1, le=MAX_TOP_K)`
Annotated[…, Query(…)]	La forma recomendada desde FastAPI 0.95 para poner validaciones; la de Query como default, la del C01, sigue funcionando.	`Annotated[int, Query(ge=1, le=20)] = 5`
Body con un modelo	Un parámetro con tipo BaseModel hace que FastAPI lea el JSON, lo valide y te pase el objeto.	`opciones: Opciones`
Request	El request crudo: headers, body en streaming y app.state. Lo usas cuando el body no es JSON, como los bytes de la foto en el C01.	`request.headers.get("content-type")`
Response como parámetro	Una respuesta temporal para poner headers o cookies; FastAPI los copia a la respuesta final.	`response.headers["X-Request-ID"] = request_id`
Tipo de retorno / response_model	Valida lo que devuelves, quita los campos de más y lo documenta. Si tu salida no cumple, es un 500: el bug es tuyo.	`-> Salida`
status_code	El código del caso exitoso, en el decorador. Por defecto es 200.	`@app.post("/v1/jobs", status_code=202)`
HTTPException	Un error que decides: código, detail y headers. Hereda del HTTPException de Starlette.	`HTTPException(503, "Triton no disponible", headers={"Retry-After": "1"})`
Depends	Inyección de dependencias: FastAPI llama la función y te pasa el resultado, una vez por request.	`Annotated[Cliente, Depends(get_triton)]`
Dependencia con yield	Abre algo antes y lo cierra después: por defecto, después de enviar la respuesta; con scope="function", antes.	`una sesión de base de datos`
lifespan + app.state	Recursos de todo el proceso: se crean antes del tráfico y se cierran al apagar.	`app.state.triton, app.state.labels`
OpenAPI y /docs	FastAPI genera /openapi.json (la descripción de tu API) y las páginas /docs y /redoc. include_in_schema=False saca una ruta de ahí.	`/metrics no aparece en el schema`

1. **Routing**
   `@app.post("/v1/classify")`

   Método + path → tu función. Sin ruta: 404; con otro método: 405.

2. **Body**
   `opciones: Opciones`

   Si la ruta declara un modelo, FastAPI lee el body entero, sin tope, y lo interpreta como JSON solo si el Content-Type es de JSON (desde 0.132, si falta, tampoco). Un JSON roto ya es 422. Con request: Request, el body lo lees tú.

3. **Parámetros y dependencias**
   `top_k: int = Query(5, ge=1, le=20)`

   Convierte y valida query, path, headers, dependencias y el body. Si algo falla: un solo 422 con la lista de todos los errores, y tu función no corre.

4. **Tu función** LO ESCRIBES TÚ
   `async def classify(...)`

   async def corre en el event loop; def, en el threadpool. Aquí lanzas HTTPException con tus códigos.

5. **Errores**
   `raise HTTPException(503, ...)`

   ExceptionMiddleware lo convierte en {"detail": ...}. Una excepción que nadie atrapa termina en 500 (ServerErrorMiddleware).

6. **Modelo de salida**
   `-> Salida`

   Valida y filtra lo que devolviste. Si no cumple el modelo: 500, porque el bug es tuyo.

7. **Serializar y enviar**
   `JSONResponse(...)`

   JSON, status_code y headers, más los del Response que recibiste como parámetro. Las dependencias con yield cierran después, salvo scope="function".

Lo que hace FastAPI con cada request, en orden. Tú escribes el paso 4; los demás los pone FastAPI (con Starlette debajo) a partir de tus tipos.

EJEMPLO REAL · EL LIFESPAN DEL C01: LO QUE SE CREA UNA VEZ POR PROCESO

PYTHON `reference/c01/gateway/app/main.py (líneas 97–116)`

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    """Recursos compartidos: se crean con el event loop ya corriendo y se cierran al apagar."""
    # Etiquetas en memoria: 500 strings, índice = id de clase del training (mismo orden que el dataset)
    app.state.labels = json.loads(Path(LABELS_PATH).read_text(encoding="utf-8"))
    # Sin este límite el threadpool (40 hilos por defecto) decodificaría 40 imágenes a la vez en un
    # pod de 1 CPU: más latencia para todas y riesgo de OOMKilled. Los demás esperan como mucho
    # SLOT_WAIT_S; si no, 503.
    app.state.cpu_slots = asyncio.Semaphore(PREPROCESS_CONCURRENCY)
    app.state.inflight = 0  # requests aceptados en este proceso; sin lock: el event loop es un solo hilo
    # UN cliente por proceso: reutiliza conexiones keep-alive (sin handshake TCP por request).
    # conn_limit = máximo de requests en vuelo hacia Triton (backpressure); conn_timeout es un
    # timeout total de aiohttp para CUALQUIER llamada: queda de respaldo (el fino va en _infer).
    # Se crea aquí y no al importar: aiohttp necesita un event loop corriendo.
    app.state.triton = triton_http.InferenceServerClient(url=TRITON_URL, conn_limit=64, conn_timeout=1.0)
    yield  # aquí la app atiende tráfico
    await app.state.triton.close()  # cierra el pool de conexiones limpiamente en el SIGTERM


app = FastAPI(title="classifier-gateway", lifespan=lifespan)  # lifespan reemplaza a on_event
```

- `@asynccontextmanager` convierte la función con `yield` en el context manager (un objeto con un «al entrar» y un «al salir») que pide `FastAPI(lifespan=…)`. Antes del `yield`: carga los 500 labels, crea el semáforo de CPU, el contador de requests en vuelo y el cliente de Triton.
- El cliente se crea aquí y no al importar el módulo: el cliente aio de Triton usa aiohttp, que exige un event loop corriendo (el hilo que atiende por turnos todo lo async). uvicorn importa la app con el loop ya andando, pero el test del C01 hace `from app import main` sin loop, y ahí fallaría con `RuntimeError: no running event loop`.

- `app.state` es un objeto donde guardas lo compartido; las rutas lo leen con `request.app.state`. Un cliente por proceso reusa las conexiones abiertas (keep-alive): no paga en cada request el saludo inicial de TCP (handshake) para abrir una nueva.
- `Path(LABELS_PATH).read_text()` es I/O que bloquea, y está bien aquí: corre una vez, antes de aceptar tráfico. Dentro de una ruta `async` sería un problema.
- Después del `yield` se cierra el pool de conexiones: corre en el apagado ordenado, cuando uvicorn ya dejó de aceptar requests. En los tests, `with TestClient(app)` también corre el lifespan.

CÓMO RECORDARLA

Tú escribes funciones con tipos; FastAPI lee, valida, llama y serializa. Entrada mala: 422. Tu `HTTPException`: su código. Salida mala o excepción suelta: 500. Lo caro, una sola vez en el lifespan y en `app.state`.

TRAMPA

Mezclar `lifespan` con `@app.on_event("startup")`: si la app tiene lifespan, los handlers de on_event no se llaman, y no hay error (solo un DeprecationWarning que casi nunca ves). Y la variante: crear el cliente de aiohttp, o el aio de Triton, al importar el módulo. Con uvicorn parece andar, porque importa la app con el event loop corriendo, pero tus tests lo importan sin loop y fallan con `RuntimeError: no running event loop`.

En el curso: M7 · FastAPI + PyTorch: el punto de partida · C01 · Serving · C01 · Archivos de referencia · M7 · Qué hace un model server (y qué no)

3 async contra hilos: event loop, async def contra def, trabajo bloqueante, threadpool, workers y GIL

¿Por qué un solo time.sleep, un requests.get o un decode dentro de async def frena a todos los requests del proceso, y dónde va cada tipo de trabajo?

SINTAXIS CON VARIABLES

```
async def funcion(...):
    datos = await cliente_async.metodo(...)   # I/O: suelta el turno mientras espera
```

Una corrutina (coroutine: lo que define `async def`) corre en el event loop, el único hilo que atiende a todos los requests `async` por turnos. Cada `await` de I/O (red, timers) le devuelve el turno al loop, que atiende a otro request mientras tanto. Entre dos `await`, tu código tiene el hilo para él solo.

```
resultado = await run_in_threadpool(funcion_cpu, args)   # Starlette/FastAPI: pool de AnyIO, 40 hilos
resultado = await asyncio.to_thread(funcion_cpu, args)   # pool de asyncio: min(32, CPUs + 4) hilos
```

Lo que bloquea (CPU, librerías sin `await`) se manda a un hilo y se espera con `await`. Son dos pools distintos: `run_in_threadpool` usa el de AnyIO (la librería async debajo de Starlette), con 40 hilos que comparten las rutas y dependencias `def`; `asyncio.to_thread` usa el grupo de hilos propio de asyncio (su executor por defecto), que no cuenta en esos 40.

```
@app.post("ruta")
def funcion(...):          # sin async: FastAPI la corre en el threadpool
    return llamada_sincrona
```

Si tu ruta usa una librería sin `await` (requests, psycopg2, boto3), declárala con `def`: FastAPI la corre en el threadpool y el event loop sigue libre. Lo mismo hace con las dependencias `def`. Un `async def` «por si acaso» con código bloqueante adentro es lo peor de los dos mundos.

```
PYTHONASYNCIODEBUG=1 comando
asyncio.get_running_loop().slow_callback_duration = segundos   # default: 0.1
```

El modo debug de asyncio avisa cada paso que tuvo el loop ocupado más de `slow_callback_duration` (100 ms por defecto), con el nombre de la corrutina culpable. Con el umbral en 20 ms, como en el lab: `Executing <Task … coro=< funcion() running at …> …> took 0.055 seconds`. Úsalo en desarrollo o en una prueba de carga (load test), no en producción.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ruta · *funcion*	el path de la ruta y la función que la atiende	`/v1/classify · classify`
cliente_async . *metodo*	un cliente con métodos async, creado una vez al arrancar (en el lifespan)	`app.state.triton.infer`
datos · *resultado*	lo que devuelve la espera	`result · batch, digest`
funcion_cpu · *args*	la función que bloquea y sus argumentos	`_decode_and_hash · data, los bytes de la foto`
llamada_sincrona	una llamada sin await	`requests.get(url, timeout=0.2).json()`
comando · *segundos*	cómo corres el proceso y desde cuánto un paso cuenta como lento	`python3.13 bloqueo.py · 0.02`
N, t_{bloqueo}	requests que bloquean el loop uno detrás de otro, y cuánto dura cada bloqueo	`3 fotos · 50 ms de decode`
λ, t_{CPU}	requests por segundo por pod y CPU por foto	`100 · 6 ms`
CPUs	núcleos que puede usar el proceso (os.process_cpu_count())	`10 en una Mac M4 → 14 hilos`

FÓRMULAS

$$t_{\text{espera del último}} \approx N \times t_{\text{bloqueo}} \qquad 3 \times 50\ \text{ms} = 150\ \text{ms} \quad (\text{/healthz espera lo mismo})$$

$$\text{hilos ocupados} = \lambda \times t_{\text{CPU}} = 100\ \frac{\text{req}}{\text{s}} \times 0.006\ \text{s} = 0.6\ \text{por pod} \quad (\text{tope del C01: 2})$$

$$\text{AnyIO (rutas def, run_in_threadpool)} = 40 \qquad \text{asyncio.to_thread} = \min\left(32,\ \text{CPUs} + 4\right)$$

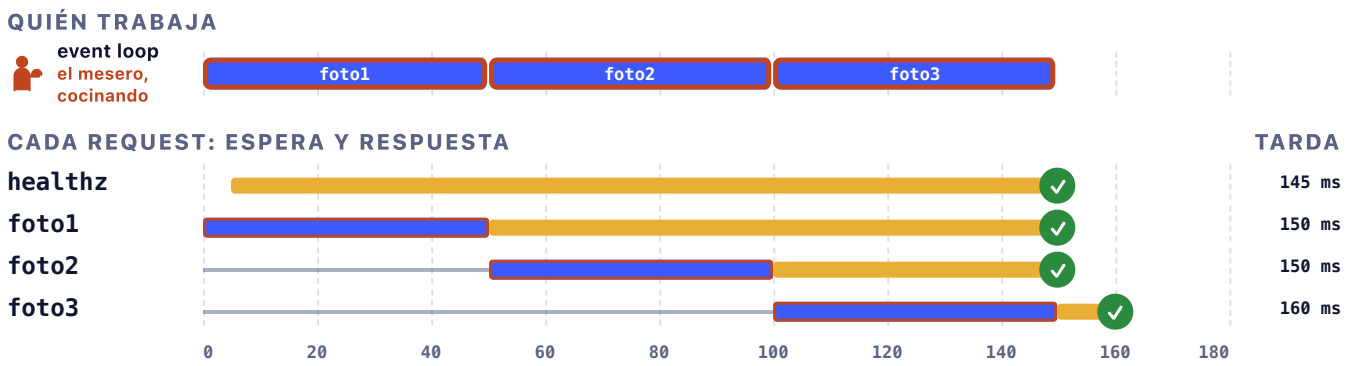
LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
event loop	El único hilo que atiende a todos los requests async del proceso, por turnos. Si un turno no termina, nadie más avanza.	`uvloop (un event loop más rápido), en el C01`
async def (corrutina)	Una función que puede pausarse en cada await y dejar pasar a las demás.	`classify, _classify, _infer`

PIEZA	QUÉ HACE	EJEMPLO
await	«Espero esto, y mientras tanto atiende a otro». Solo cede el turno si lo que esperas es async de verdad.	`await asyncio.wait_for(client.infer(...), ...)`
Llamada bloqueante	Una función que no vuelve hasta terminar y no tiene await: time.sleep, requests, open(), un decode, model(x).	`Image.open(...).convert("RGB")`
I/O-bound y CPU-bound	I/O: el tiempo se va esperando a otro (red, disco). CPU: el tiempo se va calculando. async ayuda con lo primero; para lo segundo, hilos o procesos.	`Triton es I/O para el gateway; el decode es CPU`
def en FastAPI	Las rutas y dependencias def corren en el threadpool: bloquear ahí no frena el loop.	`def metrics() en el C01`
Threadpool de AnyIO	Los 40 hilos que comparten las rutas def, las dependencias def y run_in_threadpool.	`anyio.to_thread.current_default_thread_limiter().total_tokens`
run_in_threadpool	Manda una función que bloquea al threadpool y la espera con await.	`await run_in_threadpool(_decode_and_hash, data)`
asyncio.to_thread	Lo mismo, con el pool de asyncio: min(32, CPUs + 4) hilos, aparte de los 40 de AnyIO.	`await asyncio.to_thread(decode, data)`
Semáforo	asyncio.Semaphore: un contador de turnos que deja pasar a N a la vez; el resto espera con await. Acota CPU y RAM aunque haya 40 hilos.	`cpu_slots = asyncio.Semaphore(2)`
Un hilo no se cancela	Python no puede detener código que ya corre en un hilo: si cortas la espera, el hilo sigue trabajando hasta terminar.	`por eso el C01 no le pone timeout al decode`
GIL	El candado de CPython (el intérprete normal de Python): en un proceso, un solo hilo ejecuta código Python a la vez. El código en C que lo suelta (Pillow, NumPy, hashlib) sí corre en paralelo. Desde 3.13 existe una build aparte sin GIL; la imagen del C01 usa la normal.	`el sha256 y el decode del C01, en el threadpool`
Workers (procesos)	uvicorn --workers N lanza N procesos, cada uno con su loop y su GIL. En Kubernetes se prefiere 1 por pod, y más pods.	`--workers 1 en el Dockerfile`
Modo debug de asyncio	Avisa los pasos que tuvieron el loop ocupado más de 100 ms (configurable) y dice qué corrutina fue.	`PYTHONASYNCIODEBUG=1`

Mal: el decode dentro del async def

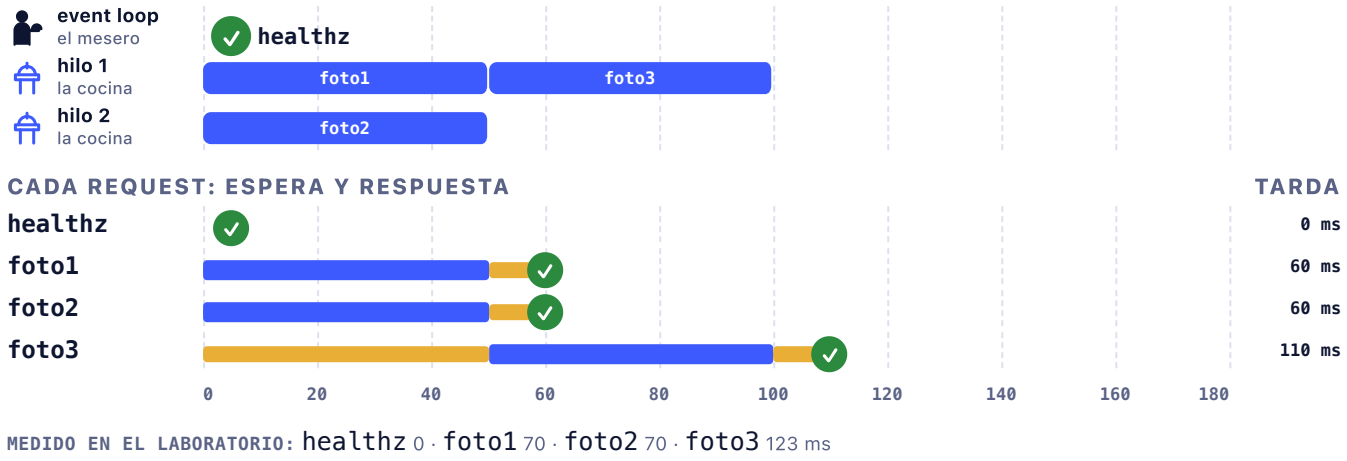
Un solo mesero (el event loop) atiende todas las mesas. Si se pone a cocinar (un decode dentro de async def), ninguna mesa avanza: ni siquiera /healthz.



MEDIDO EN EL LABORATORIO: healthz 160 · foto1 165 · foto2 165 · foto3 176 ms

Bien: el decode en el threadpool, con 2 slots

El mesero manda el plato a la cocina (los hilos del threadpool) y sigue atendiendo: /healthz responde al instante.



MEDIDO EN EL LABORATORIO: healthz 0 · foto1 70 · foto2 70 · foto3 123 ms

bloqueo.py con Python 3.13 en una Mac M4: time.sleep se pasa ≈ 5 ms en cada llamada.

CPU: el decode · CPU dentro del event loop: bloquea a todos · espera: a Triton, a un slot o a su turno · recibe la respuesta

eje en ms, a escala

Tres fotos y un /healthz en el mismo proceso: 50 ms de decode y 10 ms de espera a Triton por foto (valores ideales; en el lab, en una Mac, salen ≈ 10–20 % más). Arriba, el decode corre dentro del event loop: cada foto bloquea a las demás y /healthz espera ≈ 145 ms. Abajo, el decode va al threadpool con 2 slots, como en el C01: /healthz responde enseguida y la tercera foto espera su turno con await, sin bloquear a nadie.

EJEMPLO REAL · EL DECODE DEL C01: CPU EN UN HILO, CON UN TOPE DE 2 Y SIN TIMEOUT

PYTHON reference/c01/gateway/app/main.py (líneas 150–154 y 269–288; # … marca líneas omitidas)

```python
def _decode_and_hash(data: bytes) -> tuple[np.ndarray, str]:
    """CPU pura (bloqueante): corre en el threadpool. Pillow y numpy sueltan el GIL en decode y resize."""
    digest = hashlib.sha256(data).hexdigest()  # huella de la imagen: deduplicar y cruzar con labels
    tensor = preprocess(data, max_pixels=MAX_PIXELS)  # float32 [3,224,224], idéntico a training/eval
    return tensor[np.newaxis], digest  # batch de 1: Triton junta requests con dynamic batching
# …
    # Decode + resize son CPU pura. Una ruta `async def` que los corre directo BLOQUEA el event loop:
    # mientras decodifica una foto de 12 MP ninguna otra request avanza. En el threadpool el loop sigue
    # atendiendo I/O (otras llamadas a Triton, health checks). Máximo PREPROCESS_CONCURRENCY a la vez;
    # esperar un slot más de SLOT_WAIT_S → 503 (el pod está saturado de CPU).
    slots = request.app.state.cpu_slots
    try:
        await asyncio.wait_for(slots.acquire(), timeout=SLOT_WAIT_S)
    except TimeoutError:
        raise HTTPException(503, "gateway saturado (CPU)", headers={"Retry-After": "1"}) from None
    t = time.perf_counter()  # el reloj arranca con el slot tomado: mide CPU, no cola
    try:
        # Aquí no hay timeout a propósito: un hilo no se puede cancelar. Si cortáramos el await, el
        # slot se liberaría con el hilo todavía decodificando y el tope de RAM dejaría de valer.
        batch, digest = await run_in_threadpool(_decode_and_hash, data)
    except (ImageTooLarge, Image.DecompressionBombError) as exc:  # ImageTooLarge antes que ValueError
        raise HTTPException(413, str(exc)) from None
    except (OSError, ValueError, SyntaxError):  # lo que lanza Pillow con archivos corruptos o truncados
        raise HTTPException(400, "no se pudo decodificar la imagen") from None
    finally:
        slots.release()  # siempre: también en 400/413
```

- `_decode_and_hash` es `def`, sin `await`: CPU pura. Nunca se llama directo desde una corrutina, solo a través de `run_in_threadpool`.
- El semáforo `cpu_slots` deja 2 decodes a la vez por proceso, aunque el threadpool tenga 40 hilos: un PNG de 40 MP ocupa ≈ 320 MB al decodificarse, y el pod tiene 1 GiB.
- La espera por el slot sí tiene tope (`SLOT_WAIT_S`, 50 ms): esperar es un `await`, se puede cortar, y cortarlo no deja trabajo corriendo. Pasado el tope: 503 con `Retry-After`.
- El decode no tiene timeout a propósito: un hilo no se puede cancelar. Si cortaras el `await`, el `finally` liberaría el slot con el hilo todavía decodificando, y el tope de RAM dejaría de valer.

- El reloj de `preprocess_ms` arranca con el slot ya tomado: mide CPU, no la fila. Así el log separa «había que esperar» de «la foto era pesada».

CÓMO RECORDARLA

await para esperar, hilos para calcular, pods para escalar. Entre dos `await`, tu corrutina tiene el event loop para ella sola: si ese tramo tarda 50 ms, todo el proceso espera 50 ms.

TRAMPA

Escribir `async def` «porque es más rápido» y adentro llamar algo sin `await`: `requests.get`, `time.sleep`, un decode de Pillow o `model(x)`. Bloquea el event loop: el p99 (el tiempo de los requests más lentos) sube con la concurrencia sin que la CPU esté llena, y hasta `/healthz` y `/readyz` esperan su turno (si la espera pasa el límite de los probes, los chequeos de salud de Kubernetes, el pod sale del balanceo y hasta lo reinician). Subir `--workers` o los 40 hilos no lo arregla; sacar el bloqueo del loop, sí.

En el curso: C01 · Serving | M7 · FastAPI + PyTorch: el punto de partida | M3 · Little's law: la cuenta que ordena todo | C01 · Archivos de referencia

4 Validación y contrato: Pydantic v2, uploads con límite, 400/413/415/422, versionado y OpenAPI

¿Qué acepta y qué devuelve tu API, quién lo valida, y con qué código respondes cuando el cliente manda algo que no sirve?

SINTAXIS CON VARIABLES

```
class Modelo(BaseModel):
    model_config = ConfigDict(extra="forbid")
# un campo de más → 422
    campo: Annotated[int, Field(ge=min, le=max)] = default
```

Un modelo de Pydantic (la librería que convierte y valida datos según tus tipos) declara el JSON que aceptas. Por defecto Pydantic convierte lo razonable (`"3"` → 3: el modo lax) e ignora los campos de más; `extra="forbid"` los rechaza, y así atrapas los errores de tipeo del cliente (un campo mal escrito).

```
declared = request.headers.get("content-length", "")
if declared.isdigit() and int(declared) > MAX:
    raise HTTPException(413, "mensaje")
# sin leer un byte
size = 0
async for chunk in request.stream():
# y contando los bytes reales
    size += len(chunk)
    if size > MAX:
        raise HTTPException(413, "mensaje")
```

Así limita el C01 un body de bytes crudos: primero por el header `Content-Length` (rechazo gratis), y después contando lo que llega, porque ese header puede mentir o faltar: con `Transfer-Encoding: chunked` (el body llega en trozos, sin decir el total) no viene.

```
@app.post("/v1/recurso")
async def funcion(file: UploadFile):
    data = await file.read(MAX + 1)
# nunca más de MAX + 1 bytes en memoria
    if len(data) > MAX:
        raise HTTPException(413, "mensaje")
```

Con multipart (el formato de los formularios con archivos; necesita `python-multipart`), FastAPI recibe el archivo **entero** antes de llamar a tu función: hasta 1 MB en memoria y el resto en un archivo temporal, sin tope. Leer con tamaño máximo protege tu memoria; para cortar el upload mismo, limita el body antes, con el `RequestBodyLimitMiddleware` de Starlette (la base de FastAPI; desde 1.6) o en el proxy que está delante (por ejemplo, `client_max_body_size` en nginx).

```
v1 = APIRouter(prefix="/v1")
v2 = APIRouter(prefix="/v2")
app.include_router(v1)
app.include_router(v2)      # /v2 convive con /v1
mientras los clientes migran
```

Se versiona el contrato, no el modelo: cambiar de modelo no rompe a nadie; renombrar o quitar un campo sí, y eso es `/v2` conviviendo con `/v1`. Agregar un campo a la respuesta no rompe a nadie, siempre que los clientes ignoren los campos que no conocen.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
Modelo · *campo*	la clase del JSON que aceptas y uno de sus campos	`Opciones · top_k`
min · *max* · *default*	rango permitido y valor si no viene	`1 · 20 · 5`
MAX · *mensaje*	tope en bytes y texto del 413	`10 * 1024 * 1024 · la imagen supera 10 MB`
recurso · *funcion*	path y función del endpoint	`classify`
W, H	ancho y alto que declara el header de la imagen	`8,000 × 6,000 = 48 MP → 413`
4 B	bytes por píxel con que Pillow guarda RGB	`160 MB para 40 MP, antes de las copias`
MAX_INFLIGHT, MAX_BYTES	requests en vuelo y tope de bytes por request	`16 · 10 MB`

FÓRMULAS

$$\text{RAM al decodificar} \approx W \times H \times 4\ \text{B} \qquad 40\ \text{MP} \times 4\ \text{B} = 160\ \text{MB} \quad (\approx 320\ \text{MB de pico medido, con las copias})$$

lado corto → 256

$$1 \times 4{,}000\ \text{px} \rightarrow 256 \times 1{,}024{,}000\ \text{px} \times 4\ \text{B} \approx 1\ \text{GB}$$

$$\text{RAM de bodies} = \text{MAX_INFLIGHT} \times \text{MAX_BYTES} = 16 \times 10\ \text{MB} = 160\ \text{MB}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
400 Bad Request	El request está mal formado y el servidor no puede arreglarlo.	`body vacío; bytes que no son imagen; aspecto mayor que 1:8`
413 Content Too Large	Más grande de lo que aceptas, en bytes o en píxeles.	`más de 10 MB; más de 40 MP`
415 Unsupported Media Type	El formato (Content-Type) no es uno que proceses.	`image/gif → 415`
422 Unprocessable Content	El error de validación de FastAPI: tipos, rangos, campos que faltan o sobran. Body: {"detail": [errores]}.	`top_k=0`
Un error de Pydantic	Cada error trae type, loc (dónde), msg, input y, si aplica, ctx. FastAPI los devuelve así, sin la clave url.	`greater_than_equal en ["query", "top_k"]`
JSON roto → 422	FastAPI responde 422 con type json_invalid, no 400. Es su convención: el RFC 9110 (el estándar que define cada código HTTP) reserva el 422 para contenido bien formado.	`"msg": "JSON decode error"`
Content-Type de JSON	Desde FastAPI 0.132, un body que llega sin Content-Type de JSON (application/json o …+json) no se lee como JSON, y la validación falla con 422. Antes, si el header faltaba, FastAPI lo leía igual.	`strict_content_type=False lo desactiva`
Modo lax y modo strict	Lax, el default: convierte "3" en 3. Strict: el tipo tiene que llegar exacto.	`Field(strict=True)`
extra="forbid"	Rechaza los campos que el modelo no declara; por defecto se ignoran.	`{"top_k": 3, "debug": true} → 422 extra_forbidden`
Content-Length	El tamaño que el cliente dice que manda. Sirve para rechazar gratis, pero puede mentir o faltar.	`con chunked no viene`
request.stream()	El body por partes, a medida que llega: cortas al pasar el tope sin leerlo todo.	`_read_body_limited`
UploadFile	Un archivo de formulario multipart. FastAPI lo recibe entero antes de tu función: hasta 1 MB en memoria y el resto en disco.	`file.read(MAX + 1)`
Versión en la ruta	/v1 versiona el contrato; model_version en el body dice qué modelo respondió.	`/v1/classify`
OpenAPI	El documento JSON que describe tu API, generado de tus tipos. Lo que lees a mano (bytes) no aparece, salvo que lo declares con openapi_extra.	`/openapi.json y /docs`

↓ llega POST /v1/classify

1	¿Existe la ruta con ese método?	µs
2	¿top_k es un entero entre 1 y 20?	µs
3	¿Hay menos de 16 en vuelo?	µs
4	¿El Content-Type es jpeg, png o webp?	µs
5	¿El Content-Length dice 10 MB o menos?	µs
6	¿Los bytes reales son 10 MB o menos, y hay alguno?	1–30 ms
7	¿Queda algo de los 250 ms?	µs

no → 404 405 **FastAPI**
405 si la ruta existe con otro método; no deja línea en el gateway

no → 422 **FastAPI**
antes de tu función: lo anota el exception handler del C01

no → 503 Retry-After: 1 **classify** gateway saturado

no → 415 **_classify**

no → 413 **_classify** sin leer el body

no → 413 400 **_read_body_limited**
400 si el body vino vacío

no → 504 **_budget_left** ni siquiera decodifica

8 ¿Se libera un turno de CPU (slot) en 50 ms o menos? — 0–50 ms
→ no → 503 Retry-After: 1 _classify
gateway saturado (CPU)

9 ¿Pillow reconoce la imagen y el header dice 40 MP o menos? — header de la imagen
→ no → 400 413 preprocess.decode
400 si no es una imagen; 413 si pasa de 40 MP

10 ¿El aspecto es 1:8 o menos, y se decodifica entera? — 5–50 ms
→ no → 400 preprocess.decode
no se pudo decodificar la imagen

11 ¿Todavía queda presupuesto? — µs
→ no → 504 _budget_left
no manda a la GPU un request abandonado

12 ¿Triton respondió en min(150 ms, lo que queda)? — 5–30 ms
→ no → 504 _infer timeout de inferencia

13 ¿Sin error de red y sin rechazo de Triton?
→ no → 503 Retry-After: 1 _infer
Triton no disponible, o rechazó la inferencia

sí a todo → 200 top-k, model_version y latency_ms, con X-Request-ID

FUERA DE ESTE ÁRBOL

- 429 cuota por cliente: en el borde (una regla del WAF, el firewall de AWS), no en este main.py
- 500 una excepción que nadie atrapó: un bug
- 502 lo genera el ALB si el gateway cierra una conexión que el ALB iba a reusar

ok · rechazado a propósito: 4xx o 503 (con Retry-After, «vuelve en un momento») · falla: se acabó el tiempo o un bug

µs lo que cuesta esa revisión

Qué código devuelve el C01, en el orden en que lo decide. Cada pregunta se hace solo si las anteriores pasaron; al lado, lo que cuesta llegar hasta ahí.

EJEMPLO REAL · VALIDACIÓN EN CAPAS DEL C01: HEADER, BYTES Y PÍXELES, ANTES DE DECODIFICAR

PYTHON reference/c01/gateway/app/main.py (líneas 258–264 y 130–147) y app/preprocess.py (líneas 51–62, extracto; # … marca líneas omitidas)

```python
# gateway/app/main.py: _classify (extracto)
    ctype = request.headers.get("content-type", "").split(";")[0].strip().lower()  # sin "; charset=…"
    if ctype not in ALLOWED_TYPES:
        raise HTTPException(415, "Content-Type debe ser image/jpeg, image/png o image/webp")
    declared = request.headers.get("content-length", "")
    if declared.isdigit() and int(declared) > MAX_BYTES:  # rechazo sin leer ni un byte del body
        raise HTTPException(413, "la imagen supera 10 MB")
    data = await _read_body_limited(request)  # segunda verificación: los bytes reales

# gateway/app/main.py: _read_body_limited
async def _read_body_limited(request: Request) -> bytes:
    """Lee el body en streaming y corta en cuanto pasa de MAX_BYTES (no confía en Content-Length)."""
    chunks, size = [], 0  # acumulamos trozos para no copiar el buffer en cada iteración
    try:
        async for chunk in request.stream():  # llega por partes: podemos abortar sin leer 1 GB
            size += len(chunk)
            if size > MAX_BYTES:  # un cliente con chunked encoding o que miente en Content-Length
                raise HTTPException(413, "la imagen supera 10 MB")
            chunks.append(chunk)
    except ClientDisconnect:
        # El cliente cortó a mitad del upload (red móvil, app cerrada): no es una falla del servidor.
        # Sin esto contaría como 500 (con traceback) e inflaría ClassifierHigh5xxRate. 499 es la
        # convención de nginx ("client closed request"): queda con los 4xx, fuera de la tasa de 5xx.
        # Nadie recibe esta respuesta: uvicorn descarta lo que se manda a un cliente que ya se fue.
        raise HTTPException(499, "el cliente cerró la conexión antes de terminar el upload") from None
    if size == 0:  # body vacío: error del cliente, no vale la pena llegar al decoder
        raise HTTPException(400, "body vacío: envía los bytes de la imagen")
    return b"".join(chunks)

# gateway/app/preprocess.py: decode (extracto)
    img = Image.open(io.BytesIO(data), formats=ALLOWED_FORMATS)
    width, height = img.size  # dimensiones declaradas en el header
    # …
```

```
    if max_pixels is not None and width * height > max_pixels:
        raise ImageTooLarge(f"{width}x{height} px excede el límite de {max_pixels} px")
    if max_pixels is not None and max(width, height) > MAX_ASPECT * min(width, height):
        # …
        raise ValueError(f"relación de aspecto {width}x{height} fuera de rango (máx 1:{MAX_ASPECT})")
```

- El orden es el del costo: primero lo que está en los headers (415, y 413 por `Content-Length`), después los bytes reales a medida que llegan, y recién al final el decode. Cada rechazo cuesta menos que el paso siguiente.
- `split(";")[0].strip().lower()` acepta `image/jpeg; charset=…` o `IMAGE/JPEG`: compara solo el tipo.
- `_read_body_limited` no confía en `Content-Length`: un cliente con `Transfer-Encoding: chunked` no lo manda, y uno malicioso puede mentir. Cuenta los bytes y corta en cuanto pasa de 10 MB, sin leer 1 GB.
- Si el cliente corta a mitad del upload (red móvil, app cerrada), `request.stream()` lanza `ClientDisconnect`. El C01 lo atrapa y lo anota como **499** (la convención de nginx para «el cliente cerró la conexión»): queda con los 4xx, fuera de la tasa de 5xx, y nadie recibe la respuesta. Sin ese `except` contaría como un 500, con traceback.
- `Image.open` solo lee el header: ancho y alto salen gratis. Con eso se rechaza un PNG que declara 48 MP (413) o una tira de 1 × 4,000 px (400) antes de gastar RAM en los píxeles; `formats=` limita los decoders a JPEG, PNG y WebP.
- Ningún modelo de Pydantic valida la foto: el body son bytes crudos (no multipart: menos parsing y sin archivos temporales), así que el C01 la valida a mano. Pydantic solo valida `top_k`, y por eso `/openapi.json` no documenta el body: habría que declararlo con `openapi_extra`.

CÓMO RECORDARLA

Rechaza en el orden del costo: los headers (415, 413) antes que el body, los bytes antes que el decode (40 MP y 1:8 se leen del header de la imagen), la CPU antes que la GPU. **4xx = el cliente se equivocó**: no se reintenta igual (salvo el 429, que pide esperar el Retry-After) y no cuenta contra el SLO (la meta del servicio).

TRAMPA

Creer que un límite «ya está puesto» porque usas `UploadFile` o porque miras `Content-Length`. FastAPI recibe el upload entero antes de llamar a tu función (lo que pasa de 1 MB va a disco, sin tope), y `Content-Length` puede mentir o no venir. Con los píxeles pasa lo mismo: un PNG de 50 KB puede declarar 30,000 × 30,000 px, y el límite de bytes no lo ve.

En el curso: C01 · Serving · M11 · Feature flags y compatibilidad entre versiones · C01 · Modos de falla y degradación · M15 · Rate limiting y protección contra abuso · C01 · Archivos de referencia

5 Llamar al servidor de modelos: cliente, pool, timeouts, reintentos y deadline

¿Cómo llama tu gateway a Triton (o a cualquier servidor de modelos) sin quedarse colgado cuando el otro lado se pone lento, y cuándo vale la pena reintentar?

SINTAXIS CON VARIABLES

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.cliente =
triton_http.InferenceServerClient(
        url="host:puerto",
conn_limit=max_conexiones,
conn_timeout=respaldo_s)
    yield
    await app.state.cliente.close()
```

Un solo cliente por proceso, creado en el `lifespan` (el bloque que corre al arrancar y al apagar la app) y cerrado al final. Reutiliza conexiones **keep-alive** (conexiones TCP que quedan abiertas entre requests), así no paga el handshake (el saludo para abrir una conexión) en cada request. El **pool** es ese grupo de conexiones abiertas y `conn_limit` es su tope. La URL va sin `http://`.

```
client = httpx.AsyncClient(
    base_url="http://host:puerto",
    timeout=httpx.Timeout(por_operacion_s,
connect=conectar_s),

limits=httpx.Limits(max_connections=max_conexiones
, max_keepalive_connections=ociosas),
)
r = await client.post("ruta", content=cuerpo)
r.raise_for_status()
# al apagar, en el lifespan: await client.aclose()
```

Si tu servidor de modelos no trae cliente propio (la API HTTP de otro equipo, un servidor de modelos de lenguaje), usas **httpx**. Sus timeouts son **por operación**, de 5 s de inactividad por defecto: `connect` (abrir la conexión), `read` (esperar cada trozo de la respuesta), `write` (mandar cada trozo) y `pool` (esperar una conexión libre). No hay un tope total: ese lo pones tú, como en la plantilla siguiente. **httpx2** es su continuación, mantenida por Pydantic: la misma API con `import httpx2`.

```
deadline = t0 + presupuesto_s  # una sola vez, al
entrar
restante = deadline - time.perf_counter()
try:
    async with asyncio.timeout(min(tope_s,
restante)):
        resultado = await llamada
except TimeoutError:
    raise HTTPException(504, "mensaje") from None
```

El **deadline** es la hora límite del request: se fija una vez, al entrar. Cada salto usa lo que queda (**deadline propagation**: propagar el plazo hacia adentro). `asyncio.timeout` (Python 3.11+) y `asyncio.wait_for` cancelan la llamada al vencer y lanzan `TimeoutError`. C01 usa `wait_for`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
cliente	nombre del cliente en app.state	`triton`
host : *puerto*	dónde escucha el servidor de modelos; tritonclient lo quiere sin http://	`triton.classifier.svc.cluster.local:8000`
max_conexiones	tope de conexiones abiertas a la vez en el pool	`64 en C01; el default es 100 en tritonclient aio, aiohttp y httpx`
respaldo_s	tope total por llamada, para toda la sesión del cliente	`1.0 en C01; el default es 60.0`
por_operacion_s	tope de httpx para cada espera de read, write y pool	`1.0 (ilustrativo); el default es 5.0`
conectar_s	tope de httpx para abrir la conexión TCP	`0.2 (ilustrativo)`
ociosas	conexiones sin uso que el pool deja abiertas	`20 (default de httpx; las cierra tras 5 s sin uso)`
ruta, *cuerpo*	endpoint del servidor de modelos y los bytes que mandas	`"/v1/predict" y el JSON del request (ilustrativo)`
presupuesto_s	presupuesto total del request dentro del gateway	`0.25 (REQUEST_DEADLINE_S)`
tope_s	tope de este salto	`0.15 (INFER_TIMEOUT_S)`
llamada	la corrutina (la llamada async que esperas con await) al upstream, el servicio al que llamas	`client.infer(MODEL_NAME, [inp], outputs=[out], request_id=request_id)`
mensaje	el detail (el texto del error) que lee el caller, quien llama al gateway	`timeout de inferencia`
intentos	intentos en total, contando el primero	`2 (un reintento), solo para callers batch (scripts sin una persona esperando)`

```
for intento in range( intentos ):
    try:
        return await llamada
    except errores_transitorios :
        espera = random.uniform(0,
min( tope_espera_s , base_s * 2 ** intento))
        if intento == intentos - 1 or
time.perf_counter() + espera >= deadline:
            raise
        await asyncio.sleep(espera)
```

Reintenta solo si la operación es **idempotente** (repetirla deja el mismo efecto que hacerla una vez) y el error es **transitorio** (puede salir bien después: conexión rechazada, 503, 429). **Backoff exponencial**: la espera máxima se duplica en cada intento. **Full jitter**: la espera real cae al azar entre 0 y ese máximo, para que los clientes no vuelvan todos juntos. Si el error trae `Retry-After` (el header que dice cuántos segundos esperar), espera al menos eso; si la espera ya no cabe en el deadline, no reintentes. En el camino interactivo de C01 (una persona esperando en el formulario) nadie reintenta.

VARIABLE	QUÉ ES	EJEMPLO
errores_transitorios	excepciones que valen otro intento	con tritonclient aio (el cliente de C01): aiohttp.ClientConnectorError (no pudo conectar) o un InferenceServerException con status() "503"; con httpx: httpx.ConnectError, httpx.ConnectTimeout o un 503 o 429 convertido en excepción
base_s, *tope_espera_s*	espera base del backoff y su techo	0.1 y 2.0 (los del lab)
deadline	hora límite del request en el reloj time.perf_counter(): llegada + 250 ms	t0 + 0.25
ahora	time.perf_counter() en ese momento	t0 + 0.20 si el decode tardó 200 ms: a Triton le quedan 50 ms
L, λ, W	llamadas abiertas a la vez, llamadas por segundo en un pod, cuánto dura cada una	15 = 100 × 0.15
n	número de reintento, desde 0	0, 1, 2
$U(a,\ b)$	un número al azar entre a y b, todos igual de probables	random.uniform(0, 0.4)
A, r, c	intentos que llegan al fondo por cada request de arriba, reintentos por capa, capas que reintentan	64 = (1 + 3)³

FÓRMULAS

$$t_{\text{Triton}} = \min\left(150\text{ ms},\ \text{deadline} - \text{ahora}\right)$$

$$300\text{ ms (caller)} > 250\text{ ms (deadline)} > 150\text{ ms (Triton)} > 100\text{ ms (cola de Triton)}$$

$$L = \lambda \cdot W \qquad 100\text{ req/s} \times 0.15\text{ s} = 15\text{ llamadas abiertas por pod con Triton colgado}$$

$$\text{espera}_n \sim U\left(0,\ \min(\text{tope},\ \text{base} \cdot 2^n)\right) \qquad \text{base} = 100\text{ ms: } 100,\ 200,\ 400\text{ ms}$$

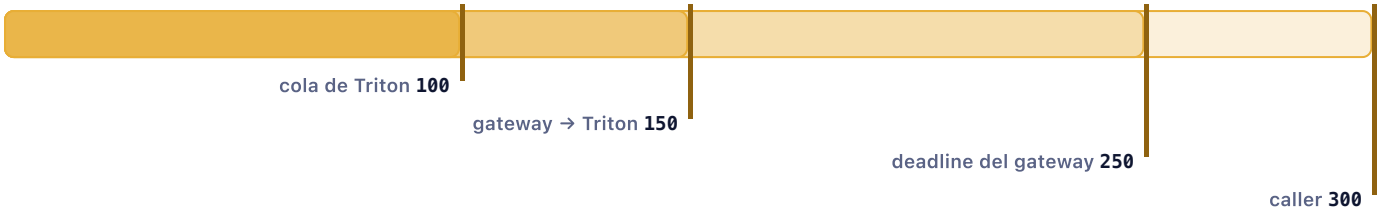
$$A = (1+r)^c \qquad (1+3)^3 = 64 \qquad (1+2)^5 = 243$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Cliente reutilizado	Uno por proceso: se crea en el lifespan y se cierra al apagar. Uno por request abre una conexión nueva cada vez; si además no lo cierras, aiohttp avisa Unclosed client session.	app.state.triton en C01
Pool de conexiones y keep-alive	Las conexiones abiertas que el cliente reutiliza. Si están todas ocupadas, la llamada espera una libre, y esa espera cuenta en el timeout.	conn_limit=64 → aiohttp.TCPConnector(limit=64); aiohttp cierra una ociosa a los 15 s

PIEZA	QUÉ HACE	EJEMPLO
HTTP o gRPC hacia Triton	C01 usa HTTP con un pool: el Service de Kubernetes (la dirección fija delante de los pods de Triton) reparte conexiones, no requests, y el pool abre varias. gRPC (el otro protocolo de Triton, con mensajes binarios) lleva todo por una sola conexión HTTP/2 de larga vida: el tráfico de un gateway quedaría pegado a un solo pod de Triton.	`con gRPC, el timeout por llamada es client_timeout, en segundos`
Timeouts por operación (httpx)	connect, read, write y pool: cada uno mide una sola espera, no el request completo. 5 s por defecto.	`ConnectTimeout, ReadTimeout, PoolTimeout`
Timeout total (aiohttp)	ClientTimeout(total=…) cubre la llamada entera, incluida la espera por una conexión del pool. Default de aiohttp: 300 s.	`conn_timeout=1.0 → ClientTimeout(total=1.0)`
Deadline y deadline propagation	Una sola hora límite por request. Cada etapa usa lo que queda y, si ya no alcanza, corta sin hacer el trabajo.	`_budget_left(deadline) → 504`
asyncio.wait_for y asyncio.timeout	Cortan cualquier await al vencer: cancelan la llamada y lanzan el TimeoutError normal de Python (desde 3.11; antes, asyncio.TimeoutError era otra clase).	`wait_for(client.infer(…), timeout=min(0.15, left))`
timeout= de infer()	No es del cliente: son microsegundos que Triton aplica en su propia cola, y solo si el modelo tiene dynamic batching (Triton junta requests en lotes).	`C01 no lo usa`
Cancelación que no llega	Cortar en el gateway libera tu conexión y tu lugar. La doc de Triton describe la cancelación de requests para su endpoint gRPC y su API de C; de HTTP no dice nada: cuenta con que el request sigue en su cola o en la GPU.	`por eso la cola de Triton rechaza a los 100 ms (REJECT)`
Idempotente	Repetirla deja el mismo efecto que hacerla una vez. POST no lo es por definición, pero clasificar una foto sí, por diseño.	`cobrar o crear un registro no lo es, salvo con una idempotency key (un id por operación con el que el servidor detecta el repetido)`
Transitorio contra permanente	Transitorio: puede salir bien después (conexión rechazada, 503, 429). Permanente: va a fallar igual (400, 413, 415, 422).	`solo los transitorios se reintentan`
Backoff exponencial con full jitter	Esperar cada vez más (el doble) y al azar dentro de ese máximo, para no volver todos en el mismo instante.	`al azar entre 0 y 100, 200 y 400 ms`
Retry budget	Un tope a la fracción de requests que pueden ser reintentos; pasado el tope, no se reintenta.	`≈ 10 % (capítulo Handling Overload del libro Site Reliability Engineering de Google)`
Amplificación de reintentos	Si cada capa reintenta por su cuenta, los intentos se multiplican: (1 + reintentos) elevado al número de capas.	`3 reintentos en 3 capas = 64 intentos al fondo; AWS cita 243 con 5 capas de 3 intentos cada una ($3^5$)`
httpx y httpx2	httpx sigue en 0.28.1 (diciembre de 2024). httpx2, de Pydantic desde mayo de 2026, es un fork (una copia que sigue por su cuenta) de esa versión: la misma API, renombrada.	`httpx2==2.13.1 en requirements-dev.txt de C01 (lo pide el TestClient, el cliente de pruebas de Starlette); hoy va en 2.16.0`

TOPES ANIDADOS, A ESCALA (DE 0 A 300 MS)



- **caller (backend de publicación)** · 300 ms
  DÓNDE: fuera del repo
  AL VENCER: la interfaz muestra el selector manual
- **deadline del gateway** · 250 ms
  DÓNDE: REQUEST_DEADLINE_S
  AL VENCER: 504 se agotó el presupuesto del request
- **gateway → Triton** · 150 ms
  DÓNDE: INFER_TIMEOUT_S, o lo que quede si es menos
  AL VENCER: 504 timeout de inferencia
- **cola de Triton** · 100 ms
  DÓNDE: default_queue_policy con REJECT
  AL VENCER: Triton rechaza → 503

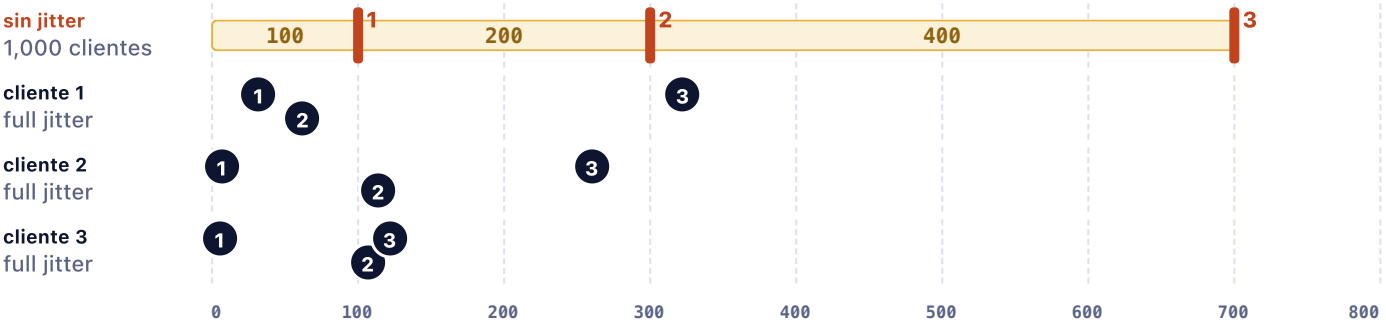
DECODE LENTO (LAB DE ESTA SECCIÓN)



decode · 200 ms Triton: le quedaban ≈ 49 ms los 150 ms que hubiera tenido Triton

504 timeout de inferencia a los ≈ 250 ms: Triton tuvo ≈ 49 ms, no 150

REINTENTOS: BACKOFF EXPONENCIAL (BASE 100 MS) Y FULL JITTER



la espera máxima: 100, 200, 400 ms, se duplica sin jitter, todos vuelven a la vez

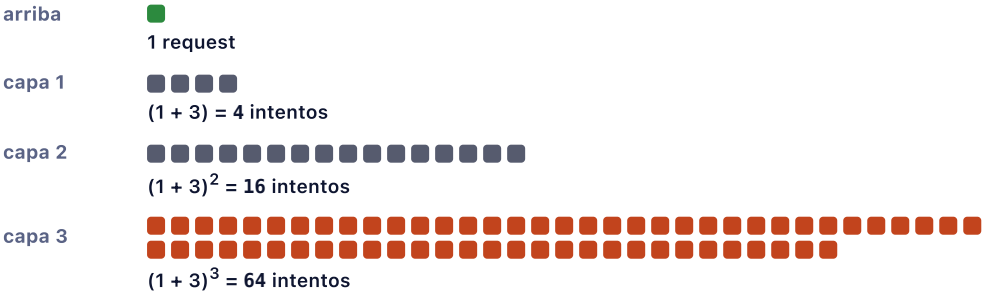
con full jitter, cada uno vuelve al azar, entre 0 y su espera máxima eje en ms, a escala

EL PICO DE 1,000 CLIENTES EN UNA VENTANA DE 10 MS



Ventanas con tráfico: 3 sin jitter contra 66 con jitter: la misma carga, repartida.

AMPLIFICACIÓN: 3 REINTENTOS EN CADA UNA DE 3 CAPAS



un tope de tiempo o una espera CPU del gateway se corta: 504 rechazado: 503

Arriba, los topes anidados de C01 a escala, en ms: cada uno cabe dentro del de afuera. En medio, un request del lab con decode lento: a Triton solo le quedan ≈ 49 ms. Abajo, 3 reintentos con backoff exponencial y full jitter: la espera máxima se duplica (100, 200 y 400 ms) y la real cae al azar dentro de ella; sin jitter, 1,000 clientes vuelven en el mismo instante.

LOS TIMEOUTS DE C01, DE AFUERA HACIA ADENTRO

TRAMO	TOPE	DÓNDE SE FIJA	AL VENCER
Caller → gateway	300 ms	backend de publicación (fuera del repo)	la interfaz muestra el selector manual de categorías
Deadline del gateway	250 ms	REQUEST_DEADLINE_S en main.py	`504` se agotó el presupuesto del request
Gateway → Triton	min(150 ms, lo que queda)	INFER_TIMEOUT_S y asyncio.wait_for en _infer	`504` timeout de inferencia
Cola de Triton	100 ms	default_queue_policy en config.pbtxt	Triton rechaza con Request timeout expired → el gateway responde `503`
Respaldo de aiohttp	1.0 s en total	conn_timeout=1.0 en el lifespan	no debería pasar: wait_for corta antes
/readyz → Triton	0.5 s	asyncio.wait_for en readyz()	no listo → `503`; con 2 fallos seguidos (≈ 10 s), el pod sale del tráfico

Cada tope es menor que el de afuera: así falla primero el de más adentro, rápido y con un código claro. El de 0.5 s de `/readyz` queda debajo del `timeoutSeconds: 2` de la probe (la pregunta periódica del kubelet).

EJEMPLO REAL · MAIN.PY: UN CLIENTE POR PROCESO Y UN DEADLINE POR REQUEST

PYTHON `reference/c01/gateway/app/main.py (líneas 41, 51–52, 107–113, 191–196, 294–306 y 169–172; # … marca líneas omitidas)`

```python
INFER_TIMEOUT_S = float(os.getenv("INFER_TIMEOUT_S", "0.15"))  # 150 ms para Triton dentro del p99 ≤ 200 ms
# …
# Presupuesto total server-side (< 300 ms del caller). Cada etapa usa lo que queda: deadline propagation
REQUEST_DEADLINE_S = float(os.getenv("REQUEST_DEADLINE_S", "0.25"))
# …
    # UN cliente por proceso: reutiliza conexiones keep-alive (sin handshake TCP por request).
    # conn_limit = máximo de requests en vuelo hacia Triton (backpressure); conn_timeout es un
    # timeout total de aiohttp para CUALQUIER llamada: queda de respaldo (el fino va en _infer).
    # Se crea aquí y no al importar: aiohttp necesita un event loop corriendo.
    app.state.triton = triton_http.InferenceServerClient(url=TRITON_URL, conn_limit=64, conn_timeout=1.0)
    yield  # aquí la app atiende tráfico
    await app.state.triton.close()  # cierra el pool de conexiones limpiamente en el SIGTERM
# …
def _budget_left(deadline: float) -> float:
    """Segundos que quedan del presupuesto. Agotado → 504: no gastes CPU ni GPU en un request abandonado."""
    left = deadline - time.perf_counter()
    if left <= 0:
        raise HTTPException(504, "se agotó el presupuesto del request")
    return left
# …
    left = _budget_left(deadline)  # no mandes a la GPU un request que el caller ya abandonó
    t = time.perf_counter()
    try:
        logits = await _infer(  # I/O: aquí el await sí cede el loop
            request.app.state.triton, batch, record["request_id"], timeout=min(INFER_TIMEOUT_S, left)
        )
    finally:
        # También si Triton falla (503) o no contesta a tiempo (504): un timeout es la llamada MÁS lenta,
        # no una que falta. Si no entrara, el p99 de inferencia se vería sano justo cuando Triton está
        # lento, y el desglose no cuadraría con la latencia total, que sí cuenta los 5xx.
        inf_s = time.perf_counter() - t
        INFERENCE.observe(inf_s)
        record["inference_ms"] = round(inf_s * 1000, 1)
# …
        result = await asyncio.wait_for(
            client.infer(MODEL_NAME, [inp], outputs=[out], request_id=request_id),
            timeout=timeout,  # ≤ INFER_TIMEOUT_S: lo que quede del presupuesto del request
        )
```

- Un cliente por proceso, creado en el `lifespan` con el event loop (el hilo que atiende las corrutinas por turnos) ya corriendo (aiohttp lo exige: fuera de un loop, su constructor falla) y cerrado con `await …close()` al apagar. `conn_limit=64` pasa a `aiohttp.TCPConnector(limit=64)`. Como el gateway acepta como mucho 16 requests a la vez, el pool nunca es el cuello.
- `conn_timeout=1.0` se vuelve `aiohttp.ClientTimeout(total=1.0)` para toda la sesión: un tope total por llamada, incluida la espera por una conexión libre. Queda de respaldo; el tope fino de cada request lo pone `asyncio.wait_for`.

- El deadline se fija una vez (`t0 + REQUEST_DEADLINE_S`). Antes del decode y antes de Triton, `_budget_left` revisa cuánto queda; si no queda nada, 504 sin gastar CPU ni GPU. A Triton le toca `min(0.15, lo que queda)` : si el decode tardó 200 ms, Triton tiene ≈ 50 ms, no 150 (el lab lo muestra).
- `asyncio.wait_for` cancela la llamada al vencer y lanza `TimeoutError` → 504. Esa cancelación corta tu lado; Triton no documenta cancelación por HTTP, así que el request puede seguir en su cola. Por eso su cola rechaza lo que lleva más de 100 ms esperando.
- La llamada se mide en un `finally` : también los 503 y los 504 dejan su tiempo en `classifier_inference_seconds` y en `inference_ms` . Un timeout es la llamada más lenta, no una que falta: si no entrara, el p99 de inferencia se vería sano justo cuando Triton está lento.
- Sin reintentos: si Triton tarda o cae, el gateway responde enseguida 504 o 503 y el caller muestra el selector manual. Los callers batch (scripts que procesan lotes, sin una persona esperando) reintentan una vez, respetando `Retry-After` , con jitter y dentro de un presupuesto de ≈ 10 %. Versiones: C01 fija tritonclient 2.72.0 y aiohttp 3.14.3; en octubre de 2026 van en 2.73.0 y 3.14.5, con la misma firma de `infer` y los mismos `conn_limit` y `conn_timeout` .

CÓMO RECORDARLA

un cliente, un pool, un deadline. El cliente se crea una vez; el pool reusa las conexiones; cada salto usa lo que queda del plazo (`300 → 250 → 150 → 100 ms`). Reintentar es pedir prestados tiempo y carga: solo si es idempotente y transitorio, con jitter, dentro del deadline y en una sola capa.

TRAMPA

confiar en los defaults. `requests` (la librería) no tiene timeout por defecto: sin `timeout` , puede esperar minutos o para siempre; httpx corta tras 5 s de inactividad en cada operación (no en total); aiohttp, a los 300 s; tritonclient aio, a los 60 s. Con un caller que se va a los 300 ms, cada llamada colgada ocupa una conexión y un lugar para una respuesta que nadie lee. Y la segunda: reintentar un timeout en el camino interactivo. Un timeout suele significar «Triton está lento por carga»: el reintento le suma carga justo ahí y tiene menos tiempo que el primer intento.

En el curso: C01 · Serving | M3 · Timeouts, reintentos y hedging | M3 · Little's law: la cuenta que ordena todo | M7 · Protocolos: REST, gRPC, KServe v2 y OpenAI-compatible | C01 · Archivos de referencia | Hoja · De PyTorch a Triton

6 Backpressure y control de admisión: límite en vuelo, 429 contra 503, load shedding

Si llegan más requests de los que tu pod puede terminar a tiempo, ¿los encolas, los rechazas o dejas que todo se caiga?

SINTAXIS CON VARIABLES

```
if request.app.state.inflight >= max_en_vuelo:
    raise HTTPException(503, "mensaje", headers={"Retry-After": "segundos"})
request.app.state.inflight += 1
try:
    resultado = await trabajo
finally:
    request.app.state.inflight -= 1
```

Control de admisión (admission control): decidir en la puerta si un request entra, antes de gastar CPU o GPU en él. Entre el `if` y el `+= 1` no hay `await`, y el event loop (un solo hilo que atiende las corrutinas por turnos) no deja que otra se meta en medio: no hace falta lock (un candado para que dos hilos no pisen el mismo dato). El `finally` devuelve el lugar aunque el trabajo falle.

```
try:
    await asyncio.wait_for(semaforo.acquire(), timeout=espera_max_s)
except TimeoutError:
    raise HTTPException(503, "mensaje", headers={"Retry-After": "segundos"}) from None
try:
    resultado = await run_in_threadpool(funcion, datos)
finally:
    semaforo.release()
```

Un **semáforo** (`asyncio.Semaphore(n)`: un contador de n lugares; `acquire` toma uno o espera, `release` lo devuelve) acota el trabajo caro. `run_in_threadpool` corre la función en un hilo del threadpool (el grupo de hilos para trabajo bloqueante). La espera con tope es una **cola acotada**: si no hay lugar en 50 ms, ya no llegas a la meta de latencia y es mejor rechazar.

```
cuota = baldes[identidad_verificada]
if not cuota.tomar():
    raise HTTPException(429, "mensaje", headers={"Retry-After": "segundos"})
```

Cuota por caller (quien llama al gateway) con un **token bucket** (un balde de fichas que se recarga a ritmo fijo; cada request gasta una). **429** = tú te pasaste de tu cuota; **503** = yo estoy lleno. La identidad sale de algo verificado (el token de servicio que valida el ALB, el balanceador de AWS, o mTLS: TLS con certificado también del lado del cliente), nunca de un header que escribe el caller. C01 lo propone, pero su `main.py` no lo trae.

```
uvicorn modulo:app --limit-concurrency n --backlog cola_tcp
```

Tope de todo el proceso de uvicorn, no de tu ruta: con n conexiones abiertas (cuentan también las keep-alive ociosas, abiertas a la espera de otro request) o n tareas, uvicorn responde 503 con el texto `Service Unavailable`, sin `Retry-After` (el header que dice cuándo volver), también a `/healthz` y `/readyz`. `--backlog` (2048 por defecto) es la cola de conexiones TCP que el sistema operativo guarda antes de que uvicorn las acepte.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
`max_en_vuelo`	requests aceptados a la vez por proceso	`16 (MAX_INFLIGHT)`
`mensaje`	texto del detail	`gateway saturado: reintenta más tarde`
`segundos`	Retry-After en segundos enteros	`1`
`trabajo`	la corrutina que hace el trabajo del request	`_classify(request, top_k, record, deadline=t0 + REQUEST_DEADLINE_S)`
`semaforo`	asyncio.Semaphore creado en el lifespan	`request.app.state.cpu_slots (asyncio.Semaphore(2))`
`espera_max_s`	cuánto se puede esperar un lugar	`0.05 (SLOT_WAIT_S)`
`funcion`, `datos`	lo que corre en el threadpool y sus datos	`_decode_and_hash, data`
`baldes`, `identidad_verificada`	un token bucket por caller, buscado por su identidad verificada	`client_id del token de servicio (propuesta; no está en C01)`
`modulo`, `n`, `cola_tcp`	la app de uvicorn, el tope de conexiones o tareas y la cola TCP	`app.main; C01 no usa --limit-concurrency; backlog 2048 (default)`
L, λ, W	requests en vuelo, llegadas por segundo a un pod, tiempo promedio adentro	`20 = 100 × 0.2`
n_{cola}, μ	requests esperando, requests que el pod termina por segundo	`100 esperando con μ = 100 req/s → 1 s de espera`
ρ	utilización: llegadas ÷ capacidad	`250 ÷ 167 ≈ 1.5 en el lab`

FÓRMULAS

$L = \lambda \cdot W$ $\quad 100\ \text{req/s} \times 0.2\ \text{s} = 20 \rightarrow \text{MAX_INFLIGHT} = 16\ \text{(tope por RAM)}$

$80\ \text{MB (Python)} + 2 \times 320\ \text{MB (decodes)} + 16 \times 10\ \text{MB (bodies)} = 880\ \text{MB} < 1\ \text{GiB}$

$W_{\text{cola}} = \dfrac{n_{\text{cola}}}{\mu}$ $\quad \dfrac{100\ \text{requests}}{100\ \text{req/s}} = 1\ \text{s} \gg 250\ \text{ms}$

$\rho = \dfrac{\lambda}{\mu} = \dfrac{250}{167} \approx 1.5 \Rightarrow$ a tiempo $\approx \dfrac{\mu}{\lambda} \approx 67\,\%$ (el lab da 68.6 %)

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Backpressure (contrapresión)	Avisar hacia atrás que estás lleno, para que quien manda baje el ritmo: colas con tope, y 503 o 429 con Retry-After.	`503 + Retry-After: 1`
Control de admisión	Decidir en la puerta si un request entra, antes de gastar CPU o GPU en él.	`inflight >= 16 → 503, sin leer el body`
Load shedding	Descartar carga a propósito: rechazar barato lo que no vas a poder terminar a tiempo.	`40 fotos a la vez: 14 × 200 y 26 × 503 rápidos (README de C01)`
Requests en vuelo (in-flight)	Los que ya entraron y todavía no terminan. Little: L = λ × W.	`≈ 3–5 por pod en el pico normal`
asyncio.Semaphore	Un contador de lugares para un recurso caro: acquire toma uno o espera; release lo devuelve.	`2 decodes a la vez por proceso`
Cola acotada	Esperar con tope de tiempo o de tamaño. Sin tope, la cola solo agrega espera hasta que todo vence.	`≤ 50 ms por un slot de CPU; ≤ 100 ms en la cola de Triton`
503 Service Unavailable	Yo, el servidor, estoy saturado o sin una dependencia, y es temporal. Puede llevar Retry-After.	`gateway saturado; Triton no disponible`
429 Too Many Requests	Tú, el caller, pasaste tu cuota de ritmo. Puede llevar Retry-After. La RFC 6585, que lo define, no dice cómo contar.	`C01 lo propone por caller; no está en main.py`
Retry-After	Cuánto esperar antes de volver: segundos enteros o una fecha HTTP.	`Retry-After: 1`
Token bucket	Un balde de B fichas que se recarga a r por segundo; cada request gasta una. Permite ráfagas de hasta B.	`r = 50/s y B = 100 (ejemplo de M15)`
--limit-concurrency (uvicorn)	Tope de todo el proceso de uvicorn: con n conexiones o tareas → 503 sin Retry-After, también a los health checks.	`C01 limita dentro de la ruta, no aquí`
Bulkhead y prioridad	Separar cargas para que una no agote la capacidad de otra: un gateway aparte para los scripts y prioridades en la cola de Triton.	`propuesta del serving de C01`
Medir la saturación	C01 cuenta los 503 por código, pero no exporta cuántos requests hay en vuelo. Un Gauge (una métrica que sube y baja) con set_function(lambda: app.state.inflight) lo daría en cada lectura de Prometheus (scrape).	`mejora posible; no está en C01`
Fallback del caller	Lo que hace quien llama cuando no hay respuesta útil.	`C01: el selector manual; con 300 ms de presupuesto no espera un Retry-After de 1 s`

llegan requests



la GPU

LO QUE MIDIÓ EL README DEL C01

16 fotos a la vez (README de C01)

200 16 503 0 rápidos p50 40 ms

40 fotos a la vez (README de C01)

200 14 503 26 rápidos p50 7 ms

LAB: 250 REQ/S CONTRA UN POD QUE TERMINA ≈ 167 REQ/S, 20 S

Cola sin tope

97.2 %

p99 de los que llegan a tiempo: **249.5 ms** pico en vuelo: **1,554** CPU gastada en respuestas tardías: **97.4 %**

Puertas de C01

68.6 % 31.4 %

p99 de los que llegan a tiempo: **90.2 ms** pico en vuelo: **16** CPU gastada en respuestas tardías: **0 %**

a tiempo rechazado rápido (503) tarde: el caller ya se fue

Las puertas de un pod del gateway de C01: 16 lugares en vuelo, 2 slots de decode con 50 ms de espera y la cola de Triton con 100 ms. Lo que no entra recibe un 503 con Retry-After: al instante en la primera puerta, tras ≤ 50 ms en la segunda y tras ≤ 100 ms en la cola de Triton. Abajo, el lab: el mismo pod con 1.5 veces su capacidad, con cola sin tope contra con puertas.

LAS TRES PUERTAS DE C01 (Y LA CUARTA QUE PROPONE)

PUERTA	DÓNDE	TOPE	SI ESTÁ LLENA	QUÉ YA SE GASTÓ
Requests en vuelo	classify(), antes de leer el body	16 por proceso	503 gateway saturado: reintenta más tarde, con Retry-After: 1	nada: microsegundos
Slots de CPU	_classify(), después de validar y leer el body	2 decodes; espera ≤ 50 ms	503 gateway saturado (CPU), con Retry-After: 1	leer el body (hasta 10 MB)
Cola de Triton	config.pbtxt (default_queue_policy)	espera ≤ 100 ms	Triton responde Request timeout expired; el gateway devuelve 503 Triton rechazó la inferencia: …, con Retry-After: 1	el decode en CPU
Cuota por caller (propuesta)	el gateway (token bucket) o una regla rate-based de AWS WAF (el firewall de aplicaciones de AWS) en el ALB	r y B por identidad (con WAF: N requests por ventana de 1, 2, 5 o 10 min)	429 con Retry-After (el WAF bloquea con 403 si no le configuras una respuesta propia con 429)	nada

La puerta más barata va primero. Todas responden rápido, con un código que el caller entiende; ninguna deja requests colgados.

EJEMPLO REAL · MAIN.PY: LAS DOS PUERTAS DEL GATEWAY

PYTHON `reference/c01/gateway/app/main.py (líneas 46–50, 102–106, 221–229 y 273–277; # … marca líneas omitidas)`

```python
# Load shedding: requests aceptados a la vez por proceso; el resto → 503 inmediato con Retry-After.
# ≈ lo que un pod completa en ≈ 200 ms (≈ 100 req/s × 0.2 s ≈ 20) y acotado por RAM: 16 bodies de
# hasta 10 MB = 160 MB, que caben en 1Gi junto a los decodes. Al pico hay ≈ 3–5 en vuelo por pod.
MAX_INFLIGHT = int(os.getenv("MAX_INFLIGHT", "16"))
SLOT_WAIT_S = 0.05  # espera máxima por un slot de CPU: si hay que esperar más, ya no cumple el SLO → 503
# …
    # Sin este límite el threadpool (40 hilos por defecto) decodificaría 40 imágenes a la vez en un
    # pod de 1 CPU: más latencia para todas y riesgo de OOMKilled. Los demás esperan como mucho
    # SLOT_WAIT_S; si no, 503.
    app.state.cpu_slots = asyncio.Semaphore(PREPROCESS_CONCURRENCY)
    app.state.inflight = 0  # requests aceptados en este proceso; sin lock: el event loop es un solo hilo
# …
    try:
        state = request.app.state
        if state.inflight >= MAX_INFLIGHT:  # fallar rápido (µs) > encolar lo que va a vencer de todos modos
            raise HTTPException(503, "gateway saturado: reintenta más tarde", headers={"Retry-After": "1"})
        state.inflight += 1
        try:
            result = await _classify(request, top_k, record, deadline=t0 + REQUEST_DEADLINE_S)
        finally:
            state.inflight -= 1  # siempre, también si _classify lanza
# …
    slots = request.app.state.cpu_slots
    try:
        await asyncio.wait_for(slots.acquire(), timeout=SLOT_WAIT_S)
    except TimeoutError:
        raise HTTPException(503, "gateway saturado (CPU)", headers={"Retry-After": "1"}) from None
```

- La primera puerta está antes de leer el body: un 503 de aquí cuesta microsegundos y no toca CPU ni GPU. El test `test_overload_is_503_with_retry_after` lo comprueba con `MAX_INFLIGHT = 0`: 503, `Retry-After: 1` y el Triton falso sin llamadas.
- `app.state.inflight` es un entero sin lock: entre el `if` y el `+= 1` no hay `await`, y el event loop corre una corrutina a la vez. Con `--workers 2` serían dos contadores, uno por proceso; C01 corre un proceso por contenedor.
- 16 sale de Little y de la RAM: ≈ 100 req/s × 0.2 s ≈ 20 en vuelo si cada request tardara 200 ms (la meta del p99), recortado a 16 porque 16 bodies de hasta 10 MB son 160 MB dentro del 1 GiB del pod. En el pico normal hay ≈ 3–5: llegar a 16 ya dice que algo va lento.
- La segunda puerta es una cola acotada: hasta 50 ms esperando uno de los 2 slots de decode (`PREPROCESS_CONCURRENCY`). Sin ella, solo frenaría la primera puerta: hasta 16 decodes a la vez en un pod de 1 CPU (el comentario habla de los 40 hilos del threadpool, el tope si tampoco hubiera primera puerta). Todas irían más lento, y en el peor caso 16 decodes de 40 MP (≈ 320 MB cada uno) no caben en 1 GiB: OOMKilled (el kernel mata el contenedor por pasarse de memoria).
- Medido en el README de C01 contra un Triton falso (un servidor que habla el mismo protocolo, KServe v2): 16 fotos a la vez → 16 × 200 (p50 ≈ 40 ms); 40 a la vez → 14 × 200 y 26 × 503 rápidos (p50 ≈ 7 ms), y ninguna pasa de 300 ms. Mientras tanto, el HPA (el autoscaler de pods de Kubernetes, que aquí escala por CPU al 60 %) agrega pods: el shedding compra ese tiempo.

CÓMO RECORDARLA

el portero dice «lleno, vuelve en un momento» (503 + `Retry-After`) en vez de dejar entrar a todos hasta que el local colapsa. Tres puertas en C01: 16 en vuelo, 2 decodes con 50 ms de espera y 100 ms en la cola de Triton. Y dos códigos: **429 = te pasaste tú**, **503 = estoy lleno yo**.

TRAMPA

«mejor encolar que rechazar». Una cola sin tope no agrega capacidad, solo espera: con 250 req/s contra un pod que termina ≈ 167, la cola crece sin parar, y en el lab el 97 % de los requests termina después del deadline, con el 97 % de la CPU gastada en respuestas que nadie espera. Con las puertas, el 69 % sale a tiempo y el resto recibe un 503 rápido. Otras dos: devolver 503 por una cuota (va 429) dispara tu alerta de disponibilidad por culpa de un caller; y usar solo `--limit-concurrency` le responde 503 también a `/healthz` , y el kubelet (el agente de Kubernetes en cada nodo) puede reiniciar el pod justo en el pico.

En el curso: M3 · Timeouts, reintentos y hedging · M3 · Utilización y colas: por qué no se planea al 100 % · M3 · Little's law: la cuenta que ordena todo · M15 · Rate limiting y protección contra abuso · C01 · Serving

7 Errores y degradación: un código por falla, la escalera de C01 y nada de detalles internos

Cuando algo falla, ¿qué código devuelves, qué texto ve el caller, qué queda en el log y qué sigue funcionando?

SINTAXIS CON VARIABLES

```
try:
    resultado = await asyncio.wait_for( llamada ,
timeout= tope_s )
except TimeoutError:
    raise HTTPException(504, " msg_lento ") from None
except  error_de_red :
    raise HTTPException(503, " msg_caido ", headers=
{"Retry-After": " s "}) from None
except  error_del_upstream :
    raise HTTPException(503, " msg_rechazo ",
headers={"Retry-After": " s "}) from None
```

Traduce en la frontera, con excepciones específicas: lento → 504; caído → 503 con `Retry-After` (cuántos segundos esperar); el upstream (el servicio al que llamas) respondió con un error → 503, como C01. Sirve para «cola llena» o «modelo cargando»; un error de contrato (el request no cumple lo que el upstream espera, como un shape equivocado) no se arregla reintentando. `from None` esconde la excepción original en el traceback (el reporte del error en el log). Lo que ve el caller (quien llama al gateway) es el `detail`, el texto del error: corto y estable.

```
@app.exception_handler( MiError )
async def  manejador (request: Request, exc:
 MiError ):
    log.warning(" evento : %r", exc)
    return JSONResponse({"detail":
" mensaje_publico "}, status_code= codigo )
```

Un solo lugar que convierte una excepción tuya en respuesta. El mensaje público es fijo; el detalle técnico (`repr(exc)`) va al log. Para cambiar cómo sale cualquier `HTTPException`, registra el handler para el de Starlette (`starlette.exceptions.HTTPException`), como indica la doc de FastAPI.

```
HTTP/1.1  codigo   frase
retry-after:  s
x-request-id:  request_id
content-type: application/json

{"detail":" mensaje_publico "}
```

Así le llega un error de C01 al caller: el código, `Retry-After` si es temporal, el id para reportar el caso y un texto corto. Starlette manda los nombres de los headers en minúsculas; HTTP no distingue. Un bug (una excepción que nadie atrapó) sale como 500 con el texto plano `Internal Server Error`: Starlette no manda el traceback mientras `debug` sea False, que es el default.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
llamada, *tope_s*	la llamada al upstream y su tope	`client.infer(…), min(0.15, lo que queda)`
error_de_red	excepción de red del cliente HTTP	`aiohttp.ClientError (tritonclient aio); httpx.TransportError (httpx; ojo: también incluye sus timeouts, httpx.TimeoutException, que no son TimeoutError: atrápalos antes, con su propio except, para responder 504)`
error_del_upstream	excepción cuando el upstream responde con un error	`InferenceServerException; httpx.HTTPStatusError después de raise_for_status()`
msg_lento, *msg_caido*, *msg_rechazo*	textos fijos para el caller	`timeout de inferencia · Triton no disponible · Triton rechazó la inferencia`
s	segundos de Retry-After	`1`
MiError, *manejador*, *evento*	tu excepción, su handler y el nombre del evento en el log	`UpstreamRechazo, manejar_rechazo, upstream_rechazo (ilustrativo)`
mensaje_publico, *codigo*, *frase*	texto para el caller, código HTTP y su frase	`gateway saturado: reintenta más tarde · 503 · Service Unavailable`
request_id	el id del request	`abc123`
5xx, 4xx, todos	requests por segundo según su código, sumados entre pods	`classifier_requests_total{status=~"5.."}`
presupuesto de err	lo que una meta de 99.9 % deja fallar en 30 días	`300,000 requests a 10 M por día`
L	requests colgados esperando a un Triton caído (Little)	`75 sin breaker; ≈ 0.5 con el breaker abierto (cálculo de M15)`

FÓRMULAS

$$\text{tasa de 5xx} = \frac{\sum \text{rate(5xx)}}{\sum \text{rate(todos)} - \sum \text{rate(4xx)}} \qquad \text{alerta urgente (page) si} > 1\ \% \text{ durante 5 min}$$

$$\text{presupuesto de error} = (1 - 0.999) \times 10\ \text{M/día} \times 30\ \text{días} = 300{,}000\ \text{requests al mes}$$

sin breaker: $L = 500 \text{ req/s} \times 0.15 \text{ s} = 75$ colgados breaker abierto: $500 \times 0.001 \text{ s} \approx 0.5$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
4xx	Culpa del request: repetirlo igual falla igual, y no cuenta contra el SLO (la meta de servicio). La excepción es 429: se reintenta después del Retry-After.	`400, 413, 415, 422; 429 sí se reintenta`
5xx	Culpa nuestra o de una dependencia: cuenta contra el SLO y gasta presupuesto de error.	`500, 503, 504`
500 Internal Server Error	Un bug: algo que nadie atrapó. Starlette responde texto plano y uvicorn loguea el traceback (Exception in ASGI application).	`C01 lo cuenta como status 500 en su línea de log`
502 Bad Gateway	Soy un gateway y el upstream me dio una respuesta inválida: rota o cortada.	`el ALB (el balanceador de AWS) lo da si uvicorn cierra una conexión ociosa que el ALB iba a reusar: --timeout-keep-alive (5 s por defecto) debe ser mayor que los 60 s del ALB; C01 usa 75`
503 Service Unavailable	Ahora no puedo, y es temporal: sobrecarga o una dependencia caída. Con Retry-After.	`gateway saturado; Triton caído o rechazando`
504 Gateway Timeout	Soy un gateway y el upstream no respondió a tiempo.	`Triton lento; presupuesto agotado`
HTTPException	La excepción de FastAPI que se vuelve respuesta: status_code, detail (lo que ve el caller) y headers.	`HTTPException(503, …, headers={"Retry-After": "1"})`
exception_handler	Una función que convierte una excepción tuya en respuesta, en un solo lugar.	`@app.exception_handler(MiError)`
debug=False	El default: un 500 no muestra el traceback. Con debug=True, Starlette lo manda en la respuesta.	`nunca debug=True en producción`
No filtrar detalles internos	Al caller, un texto corto y estable; rutas, hosts, versiones y trazas, al log.	`C01: el 413 cita las dimensiones de la imagen; el 503 de Triton lleva el texto de Triton`
Escalera de degradación	Lo que sacrificas primero, decidido antes del incidente: cada escalón conserva lo más valioso.	`6 escalones en C01`
Fallback	El camino sin el modelo: más barato y sin depender de lo que falló.	`C01: el selector manual en la interfaz del caller`
Circuit breaker	Dejar de llamar un rato a una dependencia que falla y responder el fallback al instante. Tres estados: closed (llama normal), open (no llama) y half-open (deja pasar unas llamadas de prueba).	`propuesta de M15; C01 sale del ALB con la readiness`
Falla silenciosa	Todo responde 200 y rápido, pero mal. Ningún código de error la ve.	`labels.json desalineado con el modelo`

SANO

1 Normal
CUÁNDO: todo sano

QUÉ HACE: top-5 del modelo en producción con p99 ≤ 200 ms; todos los callers atendidos
EL USUARIO VE: sugerencias de categoría
EL GATEWAY RESPONDE: 200

2 Se pierde una GPU o una zona
CUÁNDO: un nodo GPU o una zona caídos

QUÉ HACE: la otra zona absorbe el 100 % del tráfico al ≈ 42 % de su máximo medido; el autoscaler repone
EL USUARIO VE: nada
EL GATEWAY RESPONDE: 200

3 Saturación
CUÁNDO: cola de Triton alta, CPU del gateway al tope

QUÉ HACE: load shedding por prioridad: primero los callers no interactivos (su propio gateway y prioridad 2 en Triton), después cuotas estrictas
EL USUARIO VE: la app conserva su capacidad
EL GATEWAY RESPONDE: 503 + Retry-After a los scripts; 429 por cuota (propuesta)

CAÍDO

4 Latencia degradada
CUÁNDO: Triton lento

QUÉ HACE: fallar rápido: 504 a los 150 ms, sin reintentos en el camino interactivo
EL USUARIO VE: el selector manual en esos requests
EL GATEWAY RESPONDE: `504`

5 Modelo nuevo sospechoso
CUÁNDO: el canary (la versión nueva con poco tráfico) empeora la confianza, la mezcla de categorías o la aceptación

QUÉ HACE: rollback (volver a la versión anterior): segundos si sigue viva; ≈ 5–20 min si hay que volver a desplegar su imagen
EL USUARIO VE: sugerencias del modelo anterior
EL GATEWAY RESPONDE: `200` con otro model_version

6 Servicio caído
CUÁNDO: Triton caído en las dos zonas

QUÉ HACE: la interfaz muestra el selector manual; la publicación se crea igual y el camino batch la clasifica después
EL USUARIO VE: elige la categoría a mano, como antes del modelo
EL GATEWAY RESPONDE: `503` rápido, o el ALB responde `503` sin targets

Los escalones se deciden antes del incidente, no durante.

ok · rechazado a propósito (4xx o 503) · falla (500, 502, 504)

La escalera de degradación de C01, de arriba abajo: qué se sacrifica primero cuando algo falla. Cada escalón conserva lo más valioso con lo que queda; el último sigue siendo útil porque existe un camino sin el modelo.

QUÉ DEVUELVE EL GATEWAY DE C01 EN CADA FALLA (MAIN.PY)

FALLA	CÓDIGO	DETAIL QUE VE EL CALLER	HEADERS	¿REINTENTAR?
Body vacío	`400`	body vacío: envía los bytes de la imagen	X-Request-ID	No: arregla el request
Imagen corrupta o truncada, o relación de aspecto mayor a 1:8	`400`	no se pudo decodificar la imagen	X-Request-ID	No
Más de 10 MB (por Content-Length o contando los bytes)	`413`	la imagen supera 10 MB	X-Request-ID	No
Más de 40 MP (megapíxeles)	`413`	el texto de la excepción: dimensiones y límite	X-Request-ID	No
Content-Type que no es JPEG, PNG ni WebP	`415`	Content-Type debe ser image/jpeg, image/png o image/webp	X-Request-ID	No
top_k fuera de 1–20	`422`	la lista de errores de validación de FastAPI	X-Request-ID (lo agrega su exception handler: FastAPI responde antes de llamar a classify)	No
El cliente corta a mitad del upload (ClientDisconnect)	`499`	el cliente cerró la conexión antes de terminar el upload (nadie lo recibe)	X-Request-ID	Lo decide el cliente: ya se fue
16 requests en vuelo	`503`	gateway saturado: reintenta más tarde	Retry-After: 1, X-Request-ID	Sí, después de 1 s (callers batch)
Sin slot de CPU en 50 ms	`503`	gateway saturado (CPU)	Retry-After: 1, X-Request-ID	Sí, después de 1 s
Triton caído (aiohttp.ClientError)	`503`	Triton no disponible	Retry-After: 1, X-Request-ID	Sí, después de 1 s
Triton responde con error (InferenceServerException)	`503`	Triton rechazó la inferencia: … y el texto de Triton	Retry-After: 1, X-Request-ID	Sí, después de 1 s
Triton no responde a tiempo (TimeoutError)	`504`	timeout de inferencia	X-Request-ID	No en el camino interactivo
Se agotó el presupuesto de 250 ms	`504`	se agotó el presupuesto del request	X-Request-ID	No
Triton devuelve logits con NaN o inf	`500`	Triton devolvió logits no finitos (NaN o inf)	X-Request-ID	No: hay que revisar el engine
Un bug (cualquier otra excepción)	`500`	Internal Server Error, en texto plano (Starlette)	ninguno	No: hay que arreglarlo

Ni 429 ni 502: C01 no limita por caller en el código, y trata cualquier error de Triton como «ahora no» (503). Los 4xx no entran en la tasa de error ni en el histograma de latencia; los 5xx sí.

EJEMPLO REAL · MAIN.PY: CADA FALLA CON SU CÓDIGO, Y TODOS LOS ERRORES CON SU X-REQUEST-ID

PYTHON `reference/c01/gateway/app/main.py (líneas 173–180, 279–288 y 234–239; # … marca líneas omitidas)`

```python
    except TimeoutError:  # asyncio.TimeoutError es alias de TimeoutError desde Python 3.11
        # 504: Triton está vivo pero lento (cola llena, GPU saturada); el cliente cae a su fallback
        raise HTTPException(504, "timeout de inferencia") from None
    except aiohttp.ClientError:  # conexión rechazada/reseteada, DNS: Triton caído o reiniciando
        raise HTTPException(503, "Triton no disponible", headers={"Retry-After": "1"}) from None
    except InferenceServerException as exc:  # Triton respondió error: modelo no cargado, request rechazado
        detail = f"Triton rechazó la inferencia: {exc.message()}"
        raise HTTPException(503, detail, headers={"Retry-After": "1"}) from None
# …
    try:
        # Aquí no hay timeout a propósito: un hilo no se puede cancelar. Si cortáramos el await, el
        # slot se liberaría con el hilo todavía decodificando y el tope de RAM dejaría de valer.
        batch, digest = await run_in_threadpool(_decode_and_hash, data)
    except (ImageTooLarge, Image.DecompressionBombError) as exc:  # ImageTooLarge antes que ValueError
        raise HTTPException(413, str(exc)) from None
    except (OSError, ValueError, SyntaxError):  # lo que lanza Pillow con archivos corruptos o truncados
        raise HTTPException(400, "no se pudo decodificar la imagen") from None
    finally:
        slots.release()  # siempre: también en 400/413
# …
    except HTTPException as exc:
        status = exc.status_code
        record["error"] = exc.detail
        # Re-lanzamos con el X-Request-ID para que también los errores sean trazables
        headers = {**(exc.headers or {}), "X-Request-ID": request_id}
        raise HTTPException(exc.status_code, exc.detail, headers=headers) from None
```

- Cada falla de Triton tiene su código: lento (`TimeoutError`) → 504; caído (`aiohttp.ClientError`: conexión rechazada o cortada, o un nombre que no resuelve) → 503 con `Retry-After`; responde con un error (`InferenceServerException`, como el `Request timeout expired` de su cola) → 503. No usa 502: Triton contestó bien, con un error, no con una respuesta rota. Ojo: C01 manda cualquier error de Triton como 503 «ahora no», aunque algunos (un modelo que no existe, un shape equivocado) no se arreglan reintentando.
- Un engine roto (un FP16 desbordado, pesos corruptos) puede devolver NaN o inf. `_infer` lo revisa con `np.isfinite` y responde un 500 propio, con su `X-Request-ID`. Sin esa revisión, FastAPI fallaría al escribir el JSON después del `finally`: el cliente vería un 500 y la línea y la métrica dirían 200.
- Las excepciones de la imagen van de la más específica a la más general: `ImageTooLarge` hereda de `ValueError`, así que tiene que ir antes, o caería en el 400. `str(exc)` en el 413 está bien: es nuestro texto (`ImageTooLarge`) o el de Pillow (`DecompressionBombError`, sobre 80 MP), y los dos solo citan los píxeles de la imagen del caller y el límite.
- El 503 de Triton incluye `exc.message()`, el texto de Triton. Entre equipos internos, detrás de un ALB internal (un balanceador que solo se alcanza desde la red interna), ayuda a depurar; en una API pública, ese texto iría al log y el caller recibiría un mensaje fijo.
- El `except HTTPException` de `classify` anota el error en la línea de log y relanza el mismo código y `detail` con `X-Request-ID`: el caller puede reportar «el request abc123 dio 504». Un bug no pasa por aquí: sale como el 500 de Starlette, sin X-Request-ID, aunque su línea de log sí queda con status 500.
- El decode no tiene timeout, a propósito: un hilo no se puede cancelar, y cortar el `await` liberaría el slot con el hilo todavía decodificando. Por eso los límites van antes (10 MB, 40 MP, 1:8) y el `finally` devuelve el slot también en 400 y 413.

CÓMO RECORDARLA

4xx = culpa del request, no lo repitas igual (salvo 429: espera el `Retry-After`); 503 = ahora no, vuelve después; 504 = no llegó a tiempo; 500 = algo nuestro se rompió (un bug, o logits con NaN). Al caller, un texto corto y el `X-Request-ID`; el detalle, al log. Y la escalera: decide antes qué sacrificas primero, para que el último escalón siga siendo útil (en C01, elegir la categoría a mano).

TRAMPA

`detail=str(e)` dentro de un `except Exception`. El texto de una excepción cualquiera puede traer rutas, hosts internos, versiones o pedazos de datos, y además cambia de una versión a otra, así que nadie puede depender de él. Y su gemela: atrapar todo y devolver 200 con una respuesta vacía. Para el caller parece éxito, las métricas no ven ningún error y nadie se entera: una falla silenciosa hecha a mano.

En el curso: C01 · Modos de falla y degradación · M15 · Degradación con gracia: contener la falla · M15 · SLOs, error budgets y disponibilidad compuesta · C01 · Serving

8 Health, readiness y apagado ordenado

¿Cómo sabe Kubernetes que tu gateway está vivo y listo para recibir tráfico, y cómo se apaga sin cortar requests a la mitad?

SINTAXIS CON VARIABLES

```
@app.get("/healthz")
async def healthz():
    return {"status": "ok"}
```

Liveness (¿lo reinicio?): responde mientras el proceso y su event loop (el único hilo que atiende las corrutinas por turnos) respondan. Sin dependencias: si Triton cae, reiniciar el gateway no arregla nada. Si el loop se bloquea, tampoco responde a tiempo, y la probe (la pregunta periódica del kubelet, el agente de Kubernetes en cada nodo) lo nota.

```
@app.get("/readyz")
async def readyz(request: Request):
    try:
        listo = await asyncio.wait_for( chequeo ,
timeout= tope_s )
    except ( errores ):
        listo = False
    if not listo:
        raise HTTPException(503, " mensaje ")
    return {"status": "ready"}
```

Readiness (¿le mando tráfico?): 200 solo si puedes atender de verdad. Si falla, el pod sale del Service (la lista de pods que reciben tráfico) y del ALB (el balanceador de AWS), sin reiniciarse. El tope va debajo del `timeoutSeconds` de la probe, para que siempre llegue una respuesta.

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state. recurso  =  crear_recurso
    calentar
    yield  # aquí atiende requests
    await app.state. recurso .close()
```

En el `lifespan` (el bloque que corre al arrancar y al apagar), lo que va antes del `yield` corre antes de abrir el puerto: mientras tanto, las probes ven connection refused. El **warmup** (una pasada en vacío para que el primer request real no pague la inicialización) va ahí. Lo de después corre al apagar, cuando uvicorn ya terminó (o canceló) los requests en curso.

```
# Dockerfile
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0",
"--port", "8080", "--timeout-graceful-shutdown",
" drenado_s "]
# Deployment
terminationGracePeriodSeconds:  gracia_s
lifecycle:
  preStop:
    exec:
      command: ["sleep", " prestop_s "]
```

El orden del apagado: **preStop** (un paso antes de apagar: el pod sigue atendiendo mientras el ALB lo saca), **SIGTERM** (la señal de «termina») a uvicorn, **drenado** (terminar los requests en curso sin aceptar nuevos) de hasta `--timeout-graceful-shutdown` segundos, el shutdown del lifespan y la salida. Si al vencer la gracia sigue vivo, **SIGKILL** (muerte inmediata).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
chequeo	la pregunta a la dependencia	`request.app.state.triton.is_model_ready(MODEL_NAME)`
tope_s	tope de esa pregunta, menor que el timeoutSeconds de la probe	`0.5 (la probe espera 2 s)`
errores	fallas que cuentan como no listo	`TimeoutError, aiohttp.ClientError, InferenceServerException`
mensaje	detail del 503	`el modelo no está listo en Triton`
recurso, *crear_recurso*	lo que compartes entre requests y cómo se crea	`triton · triton_http.InferenceServerClient( url=TRITON_URL, conn_limit=64, conn_timeout=1.0)`
calentar	una pasada en vacío antes de aceptar tráfico (opcional)	`C01 no calienta nada en el gateway: el warmup del modelo lo hace Triton (model_warmup)`
drenado_s	cuánto espera uvicorn los requests en curso	`20`
gracia_s	plazo total del apagado, preStop incluido	`45`
prestop_s	segundos que el preStop retrasa el SIGTERM	`15`
t_{gracia}, t_{preStop}, t_{drenado}	terminationGracePeriodSeconds, el preStop, el drenado máximo de uvicorn y el colchón	`45, 15, 20 y 10 s`
t_{app}	lo que le queda a la app después del preStop	`30 s`
f, p, d	failureThreshold, periodSeconds e initialDelaySeconds de una probe	`readiness: 2 y 5 s; liveness: 3, 10 s y 5 s`
t_{detectar}	cuánto tarda en notarse una falla	`≈ 10 s para salir del tráfico; ≈ 30 s para un reinicio`
t_{arranque}	lo que tarda el lifespan en abrir el puerto	`en C01, solo leer labels.json y crear el cliente`

FÓRMULAS

$t_{\text{gracia}} \geq t_{\text{preStop}} + t_{\text{drenado}} + \text{margen} \qquad 45 = 15 + 20 + 10$

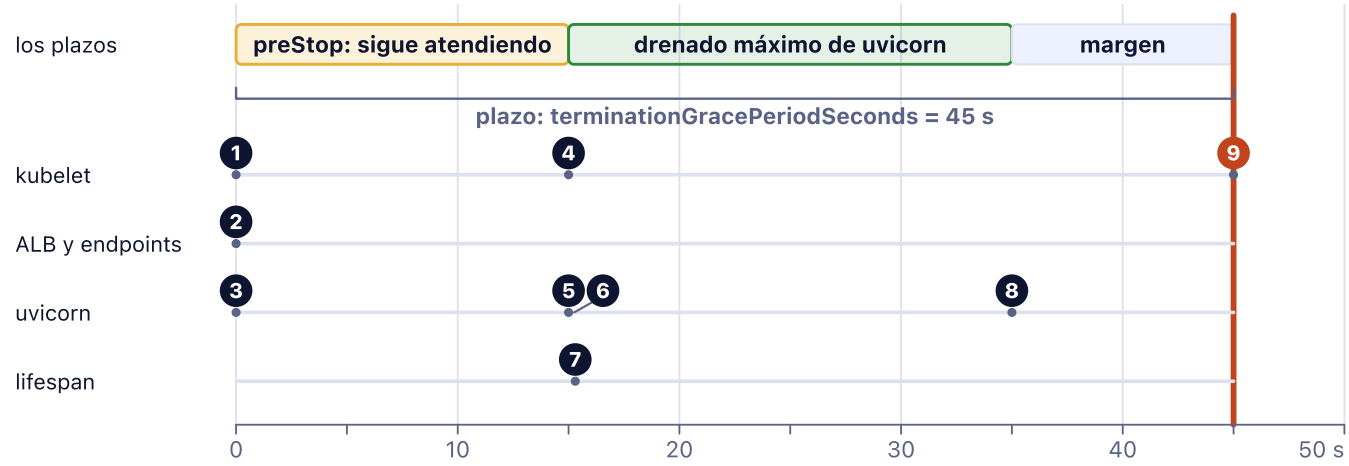
$t_{\text{app}} = t_{\text{gracia}} - t_{\text{preStop}} = 45 - 15 = 30\ \text{s} \geq 20\ \text{s de drenado}$

$t_{\text{detectar}} \approx f \times p \qquad \text{readiness: } 2 \times 5 = 10\ \text{s} \qquad \text{liveness: } 3 \times 10 = 30\ \text{s}$

$t_{\text{arranque}} < d + (f-1) \times p = 5 + 2 \times 10 = 25\ \text{s} \qquad \text{(si no cabe: startupProbe)}$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Liveness: /healthz	¿El proceso responde? Si falla 3 veces seguidas, el kubelet (el agente de Kubernetes en cada nodo) reinicia el contenedor. Corre en el event loop: si el loop está bloqueado, también falla.	`cada 10 s, timeout 2 s, 3 fallos`
Readiness: /readyz	¿Le mando tráfico? Si falla, el pod sale del Service y del ALB, sin reiniciarse.	`cada 5 s, timeout 2 s, 2 fallos`
Health check del ALB	El ALB pregunta la misma ruta y solo le manda tráfico a los pods que dan 200. Si todos fallan y siguen registrados, AWS se lo manda igual a todos (fail open); en C01 no pasa, porque los pods NotReady salen de su lista de destinos (target group).	`healthcheck-path: /readyz`
Dependencias en la readiness, nunca en la liveness	Si Triton cae, todos los gateways quedan NotReady (fuera del tráfico) y el ALB responde 503, pero nadie se reinicia en cascada.	`test_healthz_does_not_depend_on_triton`
Lifespan al arrancar	Lo que va antes del yield corre antes de abrir el puerto: las probes ven connection refused hasta que termina.	`C01: leer labels.json, el semáforo y el cliente de Triton`
Warmup	Una pasada en vacío para que el primer request real no pague la inicialización. En C01 la hace Triton (model_warmup) antes de marcar el modelo como listo.	`is_model_ready() da True después del warmup`
startupProbe	Una probe que frena a las otras dos hasta que el arranque termina. Hace falta si el arranque tarda más de lo que la liveness tolera.	`el gateway de C01 no la tiene: su lifespan no espera a nadie`
preStop	Un paso antes del SIGTERM: el gateway sigue atendiendo mientras el ALB lo saca de sus destinos (targets).	`sleep 15`
SIGTERM a uvicorn	Deja de aceptar conexiones, cierra las ociosas y espera las que tienen un request en curso.	`Shutting down`
--timeout-graceful-shutdown	Cuánto espera uvicorn a que terminen los requests en curso. Al vencer, los cancela (cada uno recibe un 500) y corre igual el shutdown del lifespan. Sin el flag, espera sin límite.	`20 s en el Dockerfile de C01`
terminationGracePeriodSeconds	Plazo total del apagado, preStop incluido. Al vencer, SIGKILL (exit 137).	`45 s en k8s/gateway.yaml`
deregistration_delay del ALB	Cuánto espera el ALB a que terminen los requests de un target que está sacando.	`30 s (el default de AWS es 300 s)`
Lifespan al apagar	Lo que va después del yield: cerrar clientes y pools. Corre después de drenar.	`await app.state.triton.close()`



1 `0 s` **kubelet** · pod borrado: Terminating; corre el preStop (sleep 15)
2 `0 s` **ALB y endpoints** · el endpoint pasa a no listo; el controller saca al pod del ALB, que espera hasta 30 s a lo que va en curso
3 `0 s` **uvicorn** · sigue atendiendo: todavía no recibió nada
4 `15 s` **kubelet** · SIGTERM al PID 1 (uvicorn)
5 `15 s` **uvicorn** · Shutting down: cierra el puerto y las conexiones ociosas; las que están en curso terminan
6 `15.25 s` **uvicorn** · en C01 el drenado suele durar < 1 s: casi todo request termina antes de su deadline de 250 ms
7 `15.3 s` **lifespan** · cierra el cliente de Triton; Application shutdown complete; sale con 0
8 `35 s` **uvicorn** · peor caso: vence --timeout-graceful-shutdown 20 y cancela lo que quede (cada uno, un 500)
9 `45 s` **kubelet** · vence la gracia: SIGKILL (exit 137)

preStop: espera a que el ALB lo saque · drenado: termina lo que está en curso · margen · SIGKILL

El apagado de un pod del gateway de C01, con t = 0 al borrar el pod. Arriba, los plazos a escala: 15 s de preStop, hasta 20 s de drenado y 10 s de margen dentro de los 45 s de gracia. Abajo, qué hace cada actor en cada momento.

EL APAGADO DE UN POD DEL GATEWAY DE C01, SEGUNDO A SEGUNDO

T	QUÉ PASA	QUIÉN
0 s	Borras el pod (o llega un rollout, el cambio a una versión nueva, o un scale-down, cuando sobran pods): queda Terminating, su endpoint pasa a no listo y empieza el preStop (sleep 15).	kubelet y control plane (los componentes que manejan el cluster)
0–15 s	El AWS Load Balancer Controller saca la IP del pod del target group (la lista de destinos del ALB): el ALB deja de mandarle requests nuevos y espera los que van en curso (deregistration_delay de 30 s). uvicorn sigue atendiendo: todavía no recibió nada.	controller, ALB y uvicorn
15 s	SIGTERM a uvicorn, que es el PID 1 (el primer proceso del contenedor): Shutting down. Cierra el puerto y las conexiones ociosas; las que tienen un request en curso se cierran al responder (Waiting for connections to close).	kubelet → uvicorn
15 s + < 1 s (casi siempre)	Los requests en curso terminan: casi todos duran menos que el deadline de 250 ms, así que el drenado real suele durar menos de un segundo. Un upload lento o un decode enorme pueden tardar más (el deadline no los corta): para eso están los 20 s.	uvicorn
enseguida	Waiting for application shutdown → el lifespan cierra el cliente de Triton → Application shutdown complete → Finished server process → sale con código 0.	uvicorn y lifespan
35 s	Peor caso: vence --timeout-graceful-shutdown 20. uvicorn cancela lo que quede (Cancel N running task(s), timeout graceful shutdown exceeded): cada request cancelado recibe un 500. Después corre igual el lifespan.	uvicorn
45 s	Vence terminationGracePeriodSeconds: SIGKILL a lo que quede (exit 137).	kubelet

Igual que en las hojas de Docker y de Kubernetes: 15 s de preStop + 20 s de drenado + 10 s de margen = 45 s. Después del preStop le quedan 30 s a la app.

EJEMPLO REAL · MAIN.PY Y GATEWAY.YAML: LAS

DOS RUTAS DE SALUD Y LAS PROBES QUE LAS LLAMAN

PYTHON `reference/c01/gateway/app/main.py (líneas 342–363) y reference/c01/k8s/gateway.yaml (líneas 29–30 y 81–102, sangría reducida; # … marca líneas omitidas)`

```python
@app.get("/healthz")
async def healthz():
    """Liveness: ¿el proceso responde? SIN dependencias: si Triton cae, reiniciar el gateway no ayuda."""
    return {"status": "ok"}


@app.get("/readyz")
async def readyz(request: Request):
    """Readiness: ¿puedo recibir tráfico? Sí solo si el modelo está listo en Triton.

    Si Triton cae, TODOS los gateways quedan NotReady, el ALB se queda sin targets y responde 503: aceptable,
    porque sin Triton el gateway no sirve de nada. Lo que nunca debes hacer es poner esta dependencia
    en liveness: el kubelet reiniciaría en cascada pods sanos.
    """
    try:
        # 0.5 s máximo: el probe del kubelet tiene su propio timeout y no queremos colgarlo
        ready = await asyncio.wait_for(request.app.state.triton.is_model_ready(MODEL_NAME), timeout=0.5)
    except (TimeoutError, aiohttp.ClientError, InferenceServerException):
        ready = False  # cualquier falla hablando con Triton = no listo
    if not ready:
        raise HTTPException(503, "el modelo no está listo en Triton")
    return {"status": "ready", "model": MODEL_NAME, "model_version": MODEL_VERSION}
# … reference/c01/k8s/gateway.yaml (líneas 29–30 y 81–102, sangría reducida):
# preStop (15 s) + --timeout-graceful-shutdown de uvicorn (20 s) + margen
terminationGracePeriodSeconds: 45
# …
# readiness: ¿mandarle tráfico? /readyz = modelo listo en Triton (ver main.py)
readinessProbe:
  httpGet:
    path: /readyz
    port: http
  periodSeconds: 5
  timeoutSeconds: 2
  failureThreshold: 2
# liveness: ¿reiniciar? /healthz NO depende de Triton: si Triton cae, reiniciar gateways no ayuda
livenessProbe:
  httpGet:
    path: /healthz
    port: http
  initialDelaySeconds: 5
  periodSeconds: 10
  timeoutSeconds: 2
  failureThreshold: 3
lifecycle:
  preStop:
    exec:
      # El ALB tarda en desregistrar el target: seguimos atendiendo 15 s antes del SIGTERM
      command: ["sleep", "15"]
```

- `/healthz` no toca a Triton: responde mientras el proceso y su event loop respondan. Si Triton cae, ningún gateway se reinicia (lo prueba `test_healthz_does_not_depend_on_triton`). Si el loop se bloquea, `/healthz` tampoco responde a tiempo y la liveness lo nota en ≈ 3 × 10 s.
- `/readyz` pregunta `is_model_ready()`, que da True solo si `/v2/models/classifier/ready` responde 200; Triton marca el modelo listo después de su `model_warmup`. Cualquier falla (timeout, red, error de Triton) cuenta como no listo → 503: el pod sale del Service y del ALB, que usa la misma ruta como health check.
- El tope de 0.5 s queda debajo del `timeoutSeconds: 2` de la probe: siempre llega una respuesta, aunque sea un 503, y no se cuenta como timeout.
- Las probes de abajo: readiness cada 5 s con 2 fallos (sale del tráfico en ≈ 10 s); liveness cada 10 s, con 3 fallos y 5 s de espera inicial. No hay startupProbe: el lifespan solo lee labels.json y crea el cliente, sin hablar con Triton, así que el puerto abre enseguida aunque Triton esté caído.
- Al apagar: preStop `sleep 15`, SIGTERM, hasta 20 s de drenado (`--timeout-graceful-shutdown 20` en el Dockerfile), el lifespan cierra el cliente de Triton y el proceso sale. 15 + 20 + 10 de margen = los 45 s de `terminationGracePeriodSeconds`. Versiones: uvicorn 0.54.0 en C01, que es también la actual.

CÓMO RECORDARLA

liveness = ¿lo reinicio?, readiness = ¿le mando tráfico? Las dependencias van en la readiness, nunca en la liveness. Al apagar: **preStop (15 s) → SIGTERM → drenado (≤ 20 s) → el lifespan cierra → sale**, todo dentro de los 45 s de gracia. Como cerrar un restaurante: apagas el letrero y sigues atendiendo mientras deja de llegar gente (preStop), atiendes a quien ya está sentado (drenado) y al final apagas la cocina (lifespan).

TRAMPA

poner a Triton en `/healthz`. Si Triton cae, el kubelet reinicia todos los gateways en cascada y ninguno arregla nada: la dependencia va en `/readyz`. Otras dos: un `lifespan` que tarda más de lo que la liveness tolera (≈ 25 s en C01) muere en bucle antes de abrir el puerto, y eso se arregla con una startupProbe, no con un initialDelaySeconds enorme; y un drenado más largo que lo que queda de gracia (`--timeout-graceful-shutdown` mayor que gracia – preStop) termina en SIGKILL con requests a la mitad.

En el curso: M6 · Los objetos que sí debes conocer · M6 · Alta disponibilidad: multi-AZ, PDB y N+1 · C01 · Modos de falla y degradación · C01 · Archivos de referencia · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Hoja · Kubernetes · Hoja · Docker

9 Observabilidad: una línea JSON por request, request ID, métricas y trazas

Cuando un request sale mal o lento, ¿cómo lo encuentras, cómo sabes en qué tramo se fue el tiempo y qué no debes guardar nunca?

SINTAXIS CON VARIABLES

```
record = {"event": "evento", "request_id":
request_id, "model_version": version }
status = 500
try:
    resultado = await trabajo
    status = 200
except HTTPException as exc:
    status = exc.status_code
    record["error"] = exc.detail
    raise
finally:
    record.update(status=status,
latency_ms=round((time.perf_counter() - t0) * 1000,
1))
    log.info(json.dumps(record, ensure_ascii=False))
```

Una línea JSON por request, escrita en el `finally`: sale también en los errores. Campos con valores, nunca la imagen ni datos personales. A stdout: en un contenedor, el log es un stream (un flujo de líneas) que guarda el runtime del contenedor (el programa que lo corre: containerd en EKS).

```
request_id = request.headers.get("x-request-id") or
uuid.uuid4().hex
response.headers["X-Request-ID"] = request_id
await client.infer(modelo, [inp], outputs=[out],
request_id=request_id)
```

Respeta el id que trae el caller (quien llama al gateway); si no trae, crea uno. Devuélvelo en la respuesta (también en los errores) y pásalo al siguiente salto: el mismo id queda en la línea de log, en la respuesta y en el request a Triton.

```
REQUESTS = Counter("prefijo_requests_total",
"ayuda", ["status", "source", "model_version"])
LATENCY =
Histogram("prefijo_request_latency_seconds",
"ayuda", ["source"], buckets=bordes)
# en el finally de cada request:
REQUESTS.labels(status=str(status), source=source,
model_version=MODEL_VERSION).inc()
LATENCY.labels(source=source).observe(elapsed)
```

Counter: un número que solo sube (cuántos requests, con qué código). **Histogram**: cuenta cuántas observaciones caen bajo cada borde (**bucket**: le="0.2" cuenta todas las de hasta 0.2 s) y guarda la suma y la cantidad. Pocos **labels** (etiquetas que parten la métrica) y con valores de una lista cerrada: nunca un id. Bordes de buckets alrededor del SLO (la meta de latencia).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
evento, *request_id*, *version*	campos fijos de la línea	`classify` · el `x-request-id` o `uuid4().hex` · `MODEL_VERSION`
trabajo	la corrutina que hace el request	`_classify(request, top_k, record, deadline=…)`
modelo	nombre del modelo en Triton	`MODEL_NAME = "classifier"`
prefijo, *ayuda*	prefijo de las métricas y su texto de ayuda	`classifier` · `Latencia total de /v1/classify`
bordes	bordes de los buckets, varios cerca del SLO	`(0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2, 0.3, 0.5, 1.0, 2.5)`
servicio	service.name de las trazas	`classifier-gateway`
fraccion	fracción de trazas nuevas que se guardan	`0.01`
collector	URL base del OpenTelemetry Collector (el servicio que recibe y reenvía las trazas)	`http://otel-collector:4318`
rutas	rutas que no se trazan (expresiones regulares separadas por comas)	`healthz,readyz,metrics`
puerto	puerto del gateway	`8080`
$\hat{q}$, r, N	cuantil estimado, posición buscada y total de observaciones	`lab: 212.8 ms con los buckets de C01`
ℓ, u, c_ℓ, c_u	bordes inferior y superior del bucket donde cae r, y los conteos acumulados en esos bordes	`0.2 y 0.3 s si el p99 está entre 200 y 300 ms`
b, v	buckets finitos del histograma y combinaciones de labels; el + 4 son el bucket +Inf, _sum, _count y _created	`11 y 4 (los valores de source)`
λ	requests por segundo	`500 en el pico de diseño`

```
histogram_quantile(0.99, sum by (le)
(rate( prefijo _request_latency_seconds_bucket[5m])))
sum(rate( prefijo _requests_total{status=~"5.."}[5m]))
/ sum(rate( prefijo _requests_total{status!~"4.."}
[5m]))
```

El **p99** (el tiempo bajo el que queda el 99 % de los requests) se calcula en Prometheus: primero se suman los buckets de todos los pods (`sum by (le)`) y después se saca el cuantil. La tasa de 5xx se mide sobre los requests válidos: los 4xx no entran ni arriba ni abajo.

```
pip install opentelemetry-distro opentelemetry-
exporter-otlp
opentelemetry-bootstrap -a install
OTEL_SERVICE_NAME= servicio \
OTEL_TRACES_SAMPLER=parentbased_traceidratio
OTEL_TRACES_SAMPLER_ARG= fraccion \
OTEL_EXPORTER_OTLP_PROTOCOL=http/protobuf
OTEL_EXPORTER_OTLP_ENDPOINT= collector \
OTEL_PYTHON_FASTAPI_EXCLUDED_URLS=" rutas " \
opentelemetry-instrument uvicorn app.main:app --host
0.0.0.0 --port puerto
```

Trazas (el recorrido de un request por los servicios) sin tocar el código: `opentelemetry-bootstrap` instala las instrumentaciones de lo que tienes (FastAPI, aiohttp) y `opentelemetry-instrument` las activa. Sale un **span** (un tramo con inicio, fin y atributos) por request, con otros chicos por cada envío y lectura (`http send`, `http receive`), y otro por la llamada a Triton; el contexto viaja en el header `traceparent`. Hace exec de uvicorn (se reemplaza por él), que sigue siendo el PID 1: el proceso que recibe el SIGTERM. C01 no lo trae: el curso propone muestrear ≈ 1 % (guardar 1 de cada 100 trazas nuevas). Desde FastAPI 0.142 (C01 fija 0.141.1), FastAPI ya crea sus spans y métricas; para mandarlos al Collector (el servicio que recibe las trazas) pide `fastapi[standard]` y `FASTAPI_OTEL_AUTO_CONFIGURE=true`.

FÓRMULAS

$$\hat{q} = \ell + (u - \ell)\frac{r - c_\ell}{c_u - c_\ell} \qquad r = 0.99 \times N$$

$$\text{series} = (b + 4) \times v = (11 + 4) \times 4 = 60 \text{ por pod}$$

$$10\text{ M requests/día} \times \approx 300\text{ B} \approx 3\text{ GB/día}$$

$$\text{trazas nuevas/s} = \lambda \times 0.01 = 500 \times 0.01 = 5$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Línea JSON por request	Un evento por request con lo necesario para investigar. Una línea = un JSON: cualquier herramienta la lee sin un parser propio.	`≈ 300–350 bytes: request_id, preprocess_ms, inference_ms, status`
stdout	En un contenedor el log es un stream: el runtime lo guarda con la hora de cada línea y un agente (Fluent Bit) lo lleva a S3, el almacenamiento de AWS.	`PYTHONUNBUFFERED=1 en el Dockerfile: cada línea sale al instante`
request_id	El id que une la línea de log, la respuesta y el request a Triton. Respeta el del caller; si no trae, uuid4.	`X-Request-ID. El ALB (el balanceador de AWS) no lo manda: agrega su propio X-Amzn-Trace-Id`
Counter	Un número que solo sube y vuelve a 0 si el proceso reinicia. Se consulta con rate(), que da cuánto sube por segundo.	`classifier_requests_total{status, source, model_version}`
Histogram	Cuenta observaciones por bucket, más _sum y _count. El p99 se calcula en Prometheus sumando los buckets de todos los pods.	`classifier_request_latency_seconds`

PIEZA	QUÉ HACE	EJEMPLO
Buckets alrededor del SLO	Los bordes deciden la precisión: con 0.15, 0.2 y 0.3 s, un p99 cerca de 200 ms sale bien; con los default, que saltan de 0.1 a 0.25 s, no.	`lab de esta sección`
Labels acotados (cardinalidad)	Cada combinación de valores de labels es una serie que Prometheus guarda en memoria (cuántas hay es la cardinalidad): solo valores de una lista cerrada.	`source: mobile, web, internal u other; nunca request_id`
Desglose por tramo	Un histograma por etapa dice dónde se fue el tiempo: CPU (preproceso) o Triton (red, cola y GPU).	`classifier_preprocess_seconds y classifier_inference_seconds`
/metrics	La ruta que Prometheus lee cada 15 s. Con 1 proceso basta el registro por defecto; con --workers N hace falta el modo multiproceso (PROMETHEUS_MULTIPROC_DIR).	`generate_latest() con CONTENT_TYPE_LATEST`
Traza y span	Una traza es el viaje de un request entre servicios; cada span es un tramo con inicio, fin y atributos.	`span del gateway → span de la llamada → spans de Triton`
traceparent y muestreo	El header estándar traceparent (especificación W3C Trace Context) lleva el id de la traza al siguiente servicio. parentbased_traceidratio guarda una fracción de las trazas nuevas y respeta la decisión de quien llama.	`OTEL_TRACES_SAMPLER_ARG=0.01`
Exemplar	Un trace_id pegado a una observación del histograma: del p99 saltas a un request concreto. Solo sale en el formato OpenMetrics (la variante del formato de texto que sí los lleva; C01 expone el otro), y Prometheus lo guarda con --enable-feature=exemplar-storage.	`LATENCY.labels(source=source).observe(t, {"trace_id": …})`
Qué no loguear	Ni la imagen, ni el EXIF (los metadatos de la foto, que pueden traer la ubicación), ni la IP, ni ids de usuario, ni tokens o el header Authorization.	`C01: solo los primeros 16 caracteres del sha256 de la imagen`

POST **/v1/classify?top_k=5** **200** **17.9 ms** REQUEST_ID `6f1d0c2b9a8e4f7d8c3b2a1908f7e6d5`

1 **Una línea de log** 355 bytes


```
{"event": "classify", "request_id": "6f1d0c2b9a8e4f7d8c3b2a1908f7e6d5", "model_version": "2026-09-01-resnet50-fp16", "source": "mobile", "image_bytes": 183204, "preprocess_ms": 8.4, "image_sha256": "9c1e5a0b7d3f2e84", "inference_ms": 6.1, "top_k": [[311, 0.8312], [54, 0.0721], [57, 0.0394], [62, 0.0188], [60, 0.0093]], "status": 200, "latency_ms": 17.9}
```

2 Métricas: 6 series suben

classifier_requests_total
{model_version="2026-09-01-resnet50-fp16",source="mobile",status="200"}
+1

classifier_request_latency_seconds_bucket
{le="0.025",source="mobile"}
+1 en este bucket y en todos los mayores (17.9 ms)

classifier_preprocess_seconds_bucket
{le="0.01"}
+1 (8.4 ms)

classifier_inference_seconds_bucket
{le="0.01"}
+1 (6.1 ms)

classifier_top1_score_bucket
{le="0.9",model_version="2026-09-01-resnet50-fp16"}
+1 (0.8312)

classifier_image_bytes_bucket
{le="250000.0"}
+1 (183,204 bytes)

3 Trazas: un árbol de spans, a escala (ms)



LO QUE UNE LOS TRES: el mismo request_id en la línea de log, el header X-Request-ID y el request a Triton

el request entero · CPU del gateway · esperando a Triton · dentro de Triton · `id` el mismo request_id

Un request de 17.9 ms y lo que deja: una línea de log, seis métricas que suben en 1 (un contador y cinco histogramas) y, si activas las trazas, un árbol de spans. Valores ilustrativos con la forma real de main.py; las trazas son la propuesta del curso.

LOGS, MÉTRICAS Y TRAZAS: QUÉ PREGUNTA RESPONDE CADA UNA

SEÑAL	RESPONDE	CUESTA	EN C01
Línea de log por request	¿Qué le pasó a este request?	bytes por request: ≈ 3 GB por día a 10 M requests	el 100 % de los requests, sin datos personales, va a S3
Métricas	¿Cuántos, qué tan rápido y cuántos errores, en total y por tramo?	series de tiempo: no crecen con el tráfico, sí con los labels	Prometheus lee /metrics cada 15 s
Trazas	¿En qué servicio y en qué tramo se fue el tiempo de este request?	spans por request: por eso se muestrean	propuesta del curso: ≈ 1 %, con el mismo request_id; el código no la trae

Se unen por ids que viajan entre ellas, pero hay que ponerlos: el `trace_id` en la línea de log y en el exemplar del histograma, o el `request_id` como atributo del span. C01 hoy solo tiene el `request_id`, que une el log, la respuesta y el request a Triton. Las alertas salen de las métricas; la investigación, de los logs y las trazas.

EJEMPLO REAL · MAIN.PY: LAS MÉTRICAS, EL REQUEST_ID Y LA LÍNEA QUE SE ESCRIBE SIEMPRE

PYTHON `reference/c01/gateway/app/main.py (líneas 63–72, 210–220 y 240–249; # … marca líneas omitidas)`

```python
# Nombres = contrato con monitoring/alerts.yml: si cambias uno aquí, rompes una alerta en silencio.
# Labels acotados: status (≈ 10 códigos) × source (4) × model_version (1–2 durante un canary)
REQUESTS = Counter(  # el cliente de Python agrega el sufijo _total; el nombre expuesto es este mismo
    "classifier_requests_total", "Requests a /v1/classify por código HTTP, caller y modelo",
    ["status", "source", "model_version"],  # status: "200", "503"… · source: mobile/web/internal/other
)
LATENCY = Histogram(  # latencia server-side; los buckets rodean al SLO (0.2 s) para que el p99 sea preciso
    "classifier_request_latency_seconds", "Latencia total de /v1/classify", ["source"],
    buckets=(0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2, 0.3, 0.5, 1.0, 2.5),
)
# …
    # Si el cliente (o un proxy) ya trae un id lo respetamos: así se traza el request de punta a punta
    request_id = request.headers.get("x-request-id") or uuid.uuid4().hex
    record = {  # la línea de log que se escribe SIEMPRE, también en errores
        "event": "classify",
        "request_id": request_id,
        "model_version": MODEL_VERSION,
```

```
        # Autodeclarado: solo sirve para partir el monitoreo (mobile/web/interno), NO es identidad.
        # Cuotas y prioridad se basan en la identidad verificada en el borde (JWT de servicio o mTLS).
        "source": request.headers.get("x-client-source", "unknown"),
    }
    status = 500  # si algo inesperado explota, se cuenta como 500
# …
    finally:
        elapsed = time.perf_counter() - t0
        source = record["source"] if record["source"] in SOURCES else "other"  # cardinalidad acotada
        # Alimenta la alerta de 5xx y los paneles por caller y por modelo (canary vs estable)
        REQUESTS.labels(status=str(status), source=source, model_version=MODEL_VERSION).inc()
        if not 400 <= status < 500:  # los 4xx son rechazos rápidos: incluirlos "mejoraría" el p99 falsamente
            LATENCY.labels(source=source).observe(elapsed)
        record.update(status=status, latency_ms=round(elapsed * 1000, 1))
        # Sin imagen cruda ni PII (ni IP ni user id): solo lo necesario para monitorear y etiquetar
        log.info(json.dumps(record, ensure_ascii=False))
```

- Los nombres de las métricas son un contrato con `monitoring/alerts.yml` (lo dice el comentario de la línea 63 de main.py): si cambias uno, una alerta se queda sin datos sin avisar. `prometheus_client` expone el contador como `classifier_requests_total` y agrega una serie `_created` (la hora en que nació cada serie) por combinación de labels (se apaga con `PROMETHEUS_DISABLE_CREATED_SERIES=True`).
- El `record` se arma al entrar y se escribe en el `finally`: sale una línea por request, también en 4xx, 503 y 504. `status = 500` por defecto: si explota algo inesperado, la línea y el contador dicen 500 aunque la excepción siga su camino.
- `source` viene de un header que escribe el caller: sirve para partir los paneles, no como identidad. Antes de usarse como label se mapea a una lista cerrada (`SOURCES`; si no, `other`): un texto libre crearía una serie nueva por cada valor distinto.
- La latencia no observa los 4xx: un rechazo de 1 ms bajaría el p99 sin que nadie esté mejor; los 503 y 504 sí entran. Con `top_k = 5` la línea pesa unos 350 bytes (355 en el ejemplo de la figura; el curso redondea a ≈ 300) y no lleva la hora: la agrega el runtime del contenedor al guardarla (`kubectl logs --timestamps` la muestra).
- El 422 de `top_k` no entra a esta función (FastAPI valida antes de llamar a `classify`): lo anota un exception handler, con su línea, su conteo y su `X-Request-ID`. Lo que no pasa por ninguna función no deja línea ni métrica: con `--no-access-log`, solo lo ve el ALB, en CloudWatch (las métricas de AWS): `HTTPCode_Target_4XX_Count` si lo respondió el gateway (un 405, por usar GET) y `HTTPCode_ELB_4XX_Count` si lo respondió el propio ALB, como el 404 de una ruta que el Ingress no enruta. Versiones: prometheus-client 0.26.0 en C01, que es también la actual; OpenTelemetry Python va en 1.45.1 (la librería base) y 0.66b1 (las instrumentaciones).

CÓMO RECORDARLA

logs para un request, métricas para todos, trazas para ver dónde se fue el tiempo, unidos por un id que pones en los tres. Una línea JSON por request, escrita en el `finally`; histogramas con buckets alrededor del SLO y labels de lista cerrada; el p99 se calcula sumando buckets, nunca promediando p99.

TRAMPA

promediar el p99 de cada pod. En el lab de esta sección, tres pods con p99 de 117, 114 y 263 ms promedian 164 ms («cumplimos 200 ms»), pero el p99 real del servicio es 206 ms: no cumple. Prometheus lo calcula bien sumando buckets (`sum by (le)`) y después el cuantil. Otras dos: poner `request_id` (o cualquier valor sin límite) como label, que crea una serie por request hasta que Prometheus se queda sin memoria; y loguear la imagen, el header `Authorization` o la IP «por si acaso».

En el curso: C01 · Monitoreo, alertas y reentrenamiento · M12 · Métricas de sistema: golden signals con detalles de ML · M12 · Prometheus, Grafana y alertas por SLO · M12 · Logging de predicciones: la materia prima del monitoreo y del etiquetado · Herramientas · Monitoreo: sistema y modelo

10 Tests: pytest, TestClient, el fake de Triton y las pruebas de carga

¿Cómo pruebas el gateway sin GPU ni Triton, incluidos los timeouts, los 503 y la carga, antes de que llegue a producción?

SINTAXIS CON VARIABLES

```
@pytest.fixture
def fixture(monkeypatch):
    monkeypatch.setattr(modulo, "nombre",
doble)
    with TestClient(app) as client:
        yield client
```

Una **fixture** prepara lo que el test necesita; lo que va después del `yield` es la limpieza. `monkeypatch` cambia un atributo solo durante ese test y lo restaura al final. `with TestClient(…)` corre el **lifespan** (el arranque y el cierre de la app); sin `with`, no corre.

```
@pytest.mark.parametrize("param,esperado",
[("valor", codigo), …])
def test_caso(fixture, param, esperado):
    doble.mode = param
    assert fixture.get("ruta").status_code ==
esperado
```

TestClient llama a la app dentro del mismo proceso, sin red ni servidor: el test es un `def` normal, sin `await`. **parametrize** corre el mismo test con varias entradas, y cada una cuenta como un test aparte.

```
@pytest.mark.anyio
async def test_caso():
    async with
AsyncClient(transport=ASGITransport(app=app),
base_url="http://test") as ac:
        r = await ac.get("ruta")
    assert r.status_code == codigo
```

Un test `async def`, para cuando el test también tiene que esperar con `await` otras llamadas async. `ASGITransport` conecta el cliente directo con la app, sin red (ASGI es la interfaz entre uvicorn y tu app). El marcador `anyio` viene con AnyIO, la librería async que FastAPI ya instala. Ojo: `AsyncClient` no corre el lifespan (para eso existe `LifespanManager`, del paquete asgi-lifespan). Se importa de `httpx`, como en la doc de FastAPI, o de `httpx2`, la continuación de httpx que mantiene Pydantic, con la misma API.

```
pytest
pytest -k "patron" -x
pytest --durations=n
pytest --lf
```

`-k` elige los tests cuyo nombre contiene el patrón; `-x` para en la primera falla; `--durations=N` lista los N más lentos; `--lf` repite solo los que fallaron la última vez.

```
locust -f locustfile --headless -u usuarios -r
rampa --run-time duracion --host url
```

Una **prueba de carga** (load test) manda muchos requests a la vez para medir el p99 (la latencia bajo la cual queda el 99 % de los requests) y encontrar dónde se satura el servicio. **Locust** simula usuarios escritos en Python; con `wait_time = constant_throughput(1)`, cada usuario intenta 1 request por segundo, como máximo: si sus respuestas tardan más de 1 s, manda menos.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
fixture	nombre de la fixture; cada test la recibe como argumento	`api`
modulo	dónde vive lo que reemplazas, tal como lo usa la app	`main.triton_http`
nombre	el atributo que cambias solo durante el test	`InferenceServerClient`
doble	la clase falsa que toma su lugar	`FakeTriton`
app	la app ASGI real, la misma que carga uvicorn	`main.app`
param, *esperado*	los argumentos que cambian en cada vuelta	`mode, expected`
valor, *codigo*	cada combinación de entrada y resultado	`("down", 503)`
caso, *ruta*	lo que prueba (sale en el reporte) y el endpoint	`readyz_reflects_triton_model_readiness, /readyz`
patron	parte del nombre de los tests que quieres correr	`triton`
n	cuántos tests lentos listar	`3`
locustfile, *usuarios*, *rampa*, *duracion*, *url*	el archivo de Locust, usuarios simulados, usuarios nuevos por segundo, duración y destino	`locustfile.py, 500, 20, 20m, https://classifier.internal (M03)`
λ	requests por segundo que genera la prueba	`500, el pico de diseño de C01`
u	usuarios simulados	`500`
r	veces por segundo que cada usuario intenta su tarea (constant_throughput)	`1`
W	lo que tarda cada respuesta; si pasa de 1/r, la tasa real cae	`≤ 0.3 s (el timeout del locustfile de M03)`
t_{test}	lo que dura el test de Triton lento	`≈ 0.15 s (medido en el laboratorio)`
t_{fake}	lo que duerme el fake en modo slow	`1 s`
T_{timeout}	el timeout del gateway hacia Triton	`0.15 s (INFER_TIMEOUT_S)`

FÓRMULAS

$$\lambda = u \cdot r \qquad 500\,\frac{\text{req}}{\text{s}} = 500\ \text{usuarios} \times 1\,\frac{\text{req}}{\text{s}} \quad \text{si } W < \frac{1}{r}$$

$$t_{\text{test}} \approx \min\left(t_{\text{fake}},\ T_{\text{timeout}}\right) = \min\left(1\ \text{s},\ 0.15\ \text{s}\right) = 0.15\ \text{s}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
pytest	Busca funciones test_* en archivos test_*.py (o *_test.py) y las corre; si un assert falla, muestra los dos valores comparados.	`pytest==9.1.1 en requirements-dev.txt`
pytest.ini	La configuración de pytest: dónde buscar tests y qué opciones usar siempre.	`pythonpath = . · testpaths = tests · addopts = -q`
@pytest.fixture	Prepara lo que el test necesita; lo que va después del yield es la limpieza.	`api: la app con el fake inyectado`
monkeypatch.setattr	Cambia un atributo solo durante un test y lo restaura al terminar, aunque el test falle.	`MAX_INFLIGHT = 0 simula un pod saturado`
@pytest.mark.parametrize	El mismo test con varias entradas; cada combinación cuenta como un test.	`21 funciones, 26 tests en C01`
TestClient	Llama a la app dentro del proceso, sin red ni servidor. Con with corre el lifespan; sin with, no lo corre, y cada request usa un event loop nuevo (el bucle que atiende el código async).	`with TestClient(main.app) as client`
httpx2	El cliente HTTP que TestClient usa por debajo desde Starlette 1.2: la continuación de httpx que mantiene Pydantic, con la misma API. Con el httpx de siempre funciona, pero avisa que está deprecado; sin ninguno de los dos, falla.	`httpx2==2.13.1 en C01`
AsyncClient + ASGITransport	Cliente async contra la app, sin red, para tests async def. No corre el lifespan.	`@pytest.mark.anyio`
ASGI directo (scope, receive, send)	Llamar a la app como lo hace uvicorn, con los mensajes escritos a mano. Sirve para lo que TestClient no puede simular: un cliente que corta a mitad del body (el mensaje http.disconnect).	`post_and_disconnect: el test del 499`
un handler de logging propio	Junta las líneas que escribe tu logger para revisarlas en el test. caplog no sirve si el logger no le pasa sus líneas al logger raíz (propagate = False), como el del C01.	`LogLines y la fixture log_lines`
REGISTRY.get_sample_value	Lee el valor de una serie de prometheus_client sin parsear /metrics. El registro es de todo el proceso: compara antes y después del request.	`classifier_inference_seconds_count sube 1 con el 504`
fake (doble de prueba)	Una clase falsa con los mismos métodos y las mismas excepciones que la real. Detecta errores de contrato que un mock genérico deja pasar.	`FakeTriton: ok, down, slow, reject y nan`
MagicMock · AsyncMock	Objetos falsos (mocks) de unittest.mock que aceptan cualquier llamada. Lo que devuelve un MagicMock no se puede esperar con await; lo que devuelven los métodos de un AsyncMock, sí.	`infer = AsyncMock(return_value=…)`
app.dependency_overrides	La forma de FastAPI de cambiar una dependencia (Depends) durante los tests. C01 no usa Depends: cambia el cliente con monkeypatch.	`app.dependency_overrides[get_client] = fake`
@pytest.mark.filterwarnings	Silencia un warning esperado en ese test, para que no tape los demás.	`ignore::PIL.Image.DecompressionBombWarning`
Locust	Prueba de carga en Python: usuarios simulados contra el gateway real; reporta el p99 por endpoint.	`-u 500 -r 20 --run-time 20m`

PIEZA	QUÉ HACE	EJEMPLO
carga abierta o cerrada	Abierta: los requests llegan a tasa fija, como los usuarios reales. Cerrada: cada usuario espera su respuesta; si el servidor se frena, la prueba también, y el p99 sale mejor que la realidad (coordinated omission).	**`abierta: k6 con constant-arrival-rate; Locust es cerrada (con un timeout corto, casi abierta)`**

QUÉ SIMULA	CÓMO	CÓDIGO	TEST (×N = CASOS DE PARAMETRIZE)
Camino feliz y contrato del tensor	FakeTriton.mode = "ok"	200	**`test_classify_ok_respects_top_k_and_contract`**
PNG y WebP, top_k por defecto	parametrize por formato	200	**`test_other_formats_and_default_top_k ×2`**
Request ID propagado	header X-Request-ID: abc123	200	**`test_request_id_is_propagated`**
Tipo no permitido	image/gif y application/octet-stream	415	**`test_unsupported_content_type_is_415`**
Más de 10 MB	por Content-Length y en 11 trozos de 1 MB	413	**`test_body_over_10mb_is_413_by_content_length · …_without_content_length`**
Decompression bomb	PNG de 1 bit de 48 MP y de 81 MP	413	**`test_decompression_bomb_is_413 ×2`**
Imagen rota, rara o vacía	basura, truncada, 1×4,000 px, body vacío	400	**`test_undecodable_image_is_400 ×2 · test_extreme_aspect_ratio_is_400 · test_empty_body_is_400`**
top_k fuera de rango	top_k=0 y top_k=500; revisa la línea y el conteo	422	**`test_top_k_out_of_range_is_422`**
Upload cortado	ASGI directo: un trozo del body y http.disconnect	499 (no cuenta como 5xx)	**`test_client_disconnect_mid_upload_is_499_not_500`**
Triton caído	mode = "down"	503 + Retry-After: 1	**`test_triton_down_is_503_with_retry_after`**
Triton rechaza	mode = "reject"	503	**`test_triton_error_is_503`**
Triton lento	mode = "slow" (duerme 1 s); el timeout entra al histograma	504	**`test_slow_triton_is_504`**
Logits con NaN	mode = "nan"	500 (también en la línea y la métrica)	**`test_non_finite_logits_are_500_in_the_log_and_the_metric_too`**
Pod saturado	monkeypatch MAX_INFLIGHT = 0	503 + Retry-After: 1	**`test_overload_is_503_with_retry_after`**
Presupuesto agotado	monkeypatch REQUEST_DEADLINE_S = 0.001	504	**`test_request_deadline_is_504`**
Liveness sin Triton	mode = "down"	200	**`test_healthz_does_not_depend_on_triton`**
Readiness	ok, down y reject	200 · 503 · 503	**`test_readyz_reflects_triton_model_readiness ×3`**
Métricas	GET /metrics sin seguir redirects	200	**`test_metrics_exposes_contract_names`**

Los 26 tests del C01 y lo que simula cada uno. El fake y monkeypatch cubren cada código de error sin GPU; la carga se prueba aparte.

EJEMPLO REAL · C01: EL FAKE DE TRITON, LA FIXTURE Y EL TEST DEL 504

PYTHON **reference/c01/gateway/tests/test_api.py (líneas 38–65, 79–85 y 245–253; # … marca líneas omitidas)**

```python
class FakeTriton:
    """Doble de tritonclient.http.aio.InferenceServerClient con cinco modos de comportamiento."""

    mode = "ok"  # ok | down (sin red) | slow (no responde a tiempo) | reject (Triton devuelve error) | nan

    def __init__(self, url, conn_limit=100, conn_timeout=60.0, **kwargs):  # misma firma que el real
        self.url = url
        self.calls = []  # guardamos lo que el gateway envía para verificar el contrato del tensor
```

```
    async def infer(self, model_name, inputs, outputs=None, request_id="", **kwargs):
        self.calls.append({"model": model_name, "inputs": inputs, "outputs": outputs})
        if self.mode == "down":
            raise aiohttp.ClientConnectionError("connection refused")  # lo que se ve si el pod de Triton murió
        if self.mode == "slow":
            await asyncio.sleep(1.0)  # más que INFER_TIMEOUT_S (0.15 s): el gateway debe cortar antes
        if self.mode == "reject":
            raise InferenceServerException("Request for unknown model", status="400")
        logits = np.linspace(-2, 2, 500, dtype=np.float32)[np.newaxis]  # [1,500], el shape real
        logits[0, TOP_CLASS] = np.nan if self.mode == "nan" else 12.0  # top-1 claro, o un engine roto
        return FakeResult(logits)

    async def is_model_ready(self, model_name, model_version=""):
        if self.mode == "down":
            raise aiohttp.ClientConnectionError("connection refused")
        return self.mode != "reject"  # "reject" simula un modelo que no cargó

    async def close(self):
        pass  # el real cierra el pool de aiohttp

# …
@pytest.fixture
def api(monkeypatch):
    """App con el fake inyectado. `with TestClient` ejecuta el lifespan (crea y cierra el cliente)."""
    FakeTriton.mode = "ok"
    monkeypatch.setattr(main.triton_http, "InferenceServerClient", FakeTriton)
    with TestClient(main.app) as client:
        yield client

# …
def test_slow_triton_is_504(api, log_lines):
    FakeTriton.mode = "slow"
    observed = REGISTRY.get_sample_value("classifier_inference_seconds_count")
    r = post(api, image_bytes())
    assert r.status_code == 504  # cortamos a los 0.15 s: el cliente cae a su fallback manual
    assert "X-Request-ID" in r.headers  # los errores también son trazables
    # El timeout entra al histograma de inferencia y a la línea: es la llamada más lenta, no una que falta
    assert REGISTRY.get_sample_value("classifier_inference_seconds_count") == observed + 1
    assert log_lines[-1]["inference_ms"] >= 100  # ≈ 150 ms: lo que se esperó antes de cortar
```

- `monkeypatch` reemplaza la **clase** `InferenceServerClient`, no un objeto: el lifespan del C01 sigue llamando `triton_http.InferenceServerClient(url=TRITON_URL, conn_limit=64, conn_timeout=1.0)` y recibe un FakeTriton. Por eso el fake copia la firma del real (`conn_limit=100`, `conn_timeout=60.0` son los defaults de `tritonclient.http.aio`).
- Cada modo imita una falla real con la misma excepción que lanzaría el cliente: `down` → `aiohttp.ClientConnectionError` (503), `slow` → no responde a tiempo (504), `reject` → `InferenceServerException` (503), y `nan` → logits con NaN, como un engine roto (500). Así se prueba el mapeo de errores a códigos sin GPU.
- `slow` duerme con `await asyncio.sleep(1.0)`, que cede el **event loop** (el único hilo que atiende todas las requests `async` por turnos): el `asyncio.wait_for` del gateway puede cortar a los 0.15 s. Con `time.sleep(1.0)` no podría: el request respondería 200 al segundo y el test fallaría (lo provocas en el laboratorio).
- El test del 504 revisa también lo que queda escrito: `REGISTRY.get_sample_value` lee el conteo de `classifier_inference_seconds` (el timeout entra al histograma), y `log_lines`, una fixture con un handler de logging propio, junta las líneas JSON: el logger del C01 no propaga al raíz, así que `caplog` no las ve.
- `mode` es un atributo de la **clase**, compartido por todos los tests: la fixture lo vuelve a `"ok"` antes de cada uno. Sin esa línea, el resultado dependería del orden de los tests.
- `self.calls` guarda lo que el gateway mandó: el primer test revisa el nombre, la shape y el dtype del tensor (`input`, `[1, 3, 224, 224]`, `FP32`), el contrato de `config.pbtxt`. Un MagicMock aceptaría cualquier cosa y no lo notaría.
- El 499 no se prueba con `TestClient`, que siempre entrega el body completo. `post_and_disconnect` llama a la app como uvicorn (ASGI directo: `scope`, `receive`, `send`): manda un trozo del body y después `http.disconnect`, el mismo mensaje que llega cuando el cliente cierra la conexión.
- Lo que estos tests no ven: la red, uvicorn, el ALB y la carga. Para eso están un **smoke test** (una prueba corta de punta a punta) en staging, un entorno igual a producción, y una prueba de carga con fotos reales (M03).

CÓMO RECORDARLA

un fake por dependencia, un test por código. El fake imita al cliente real (mismos métodos, mismas excepciones), `monkeypatch` lo mete, `with TestClient` arranca la app con su lifespan, y cada código de error (400, 413, 415, 422, 499, 500, 503, 504) tiene su test.

TRAMPA

`TestClient(app)` sin `with`. El lifespan no corre, `app.state` queda vacío y el primer request falla con `AttributeError: 'State' object has no attribute 'inflight'`. Además, cada request corre en un event loop nuevo: un cliente o un semáforo creado al importar el módulo puede terminar en `Event loop is closed` o en `… is bound to a different event loop`. La otra trampa: un fake que duerme con `time.sleep`. Bloquea el loop, el timeout no corta y el test del 504 falla, o pasa por el motivo equivocado.

En el curso: C01 · Archivos de referencia · C01 · Serving · M10 · CI/CD para modelos: la pirámide de tests · M3 · Load testing: cómo se mide de verdad · Hoja · Git y GitHub Actions

11 Rendimiento: workers de uvicorn, CPU para el decode, keep-alive, perfilar y medir el p99

¿Dónde se va el tiempo de tu gateway y qué perilla mueves: hilos, procesos, pods o keep-alive?

SINTAXIS CON VARIABLES

```
uvicorn modulo:app --host 0.0.0.0 --port puerto
--workers n \
  --timeout-keep-alive seg_ka --no-access-log
--timeout-graceful-shutdown seg_apagado
```

Así arranca el C01. `--workers` son **procesos** completos, cada uno con su event loop (el hilo que atiende los requests async por turnos), su memoria y sus métricas; en Kubernetes va 1 por pod y escalas con pods. `--host 0.0.0.0` escucha en todas las interfaces de red: con el default, 127.0.0.1, nadie de afuera del contenedor llega. `--timeout-keep-alive` dice cuántos segundos queda abierta una conexión sin requests, lista para reusarse.

```
await asyncio.wait_for( slots .acquire(),
timeout= espera )
try:
    resultado = await
run_in_threadpool( funcion_cpu , datos )
finally:
    slots .release()
```

El trabajo de CPU (decode, resize) va a un **threadpool** (hilos aparte del event loop). El **semáforo** (un contador de turnos, creado en el lifespan) deja pasar N a la vez: acota CPU y RAM. Si no hay turno en `espera`, C01 responde 503.

```
py-spy top --pid pid
py-spy dump --pid pid
py-spy record -o archivo .svg --pid pid --duration
seg
```

py-spy es un profiler por muestreo: mira la pila de llamadas (stack) del proceso muchas veces por segundo, sin tocar tu código. `top` muestra las funciones que más tiempo usan, en vivo; `dump`, dónde está cada hilo ahora mismo; `record`, un **flame graph** (un gráfico de dónde se fue el tiempo). En macOS pide `sudo`; en un contenedor, el permiso (capability) SYS_PTRACE, que deja leer la memoria de otro proceso.

```
PYTHONASYNCIODEBUG=1 uvicorn modulo:app --port
puerto --loop asyncio
```

Modo debug de asyncio: avisa cada vez que algo ocupa el event loop más de 100 ms, con la corrutina y la línea donde quedó pausada después (`Executing <Task …> took 0.182 seconds`): lo que bloqueó está justo antes de esa línea. `--loop asyncio` usa el loop estándar de Python en vez de uvloop. Solo en local o en staging: el modo debug hace todo más lento.

```
curl -s -o /dev/null -w '%{http_code} %
{time_connect} %{time_starttransfer} %
{time_total}\n' \
  -H "Content-Type: image/jpeg" --data-binary
@{ foto } " url /v1/classify"
```

Mide un request desde afuera, en segundos: la conexión, el primer byte de la respuesta (incluye la subida del body y lo que tardó el servidor) y el total. Un request no da un p99: para eso, el histograma o una prueba de carga.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
`modulo` : `app`	el módulo y la variable de la app	`app.main:app`
`puerto`	puerto donde escucha	`8080`
`n`	procesos de uvicorn	`1 (más capacidad = más pods)`
`seg_ka`	segundos que una conexión ociosa queda abierta	`75 (default 5)`
`seg_apagado`	segundos de drenado tras el SIGTERM	`20`
`slots` , `espera`	el semáforo del lifespan y cuánto esperar un turno	`app.state.cpu_slots (2 turnos), SLOT_WAIT_S = 0.05`
`funcion_cpu` , `datos`	la función bloqueante y su entrada	`_decode_and_hash, data`
`pid`	el proceso a mirar, por su número (PID)	`el PID de uvicorn (1 dentro del contenedor)`
`archivo` , `seg`	dónde guardar el flame graph y cuánto muestrear	`perfil, 30`
`foto` , `url`	una imagen de prueba y el gateway	`foto.jpg, http://localhost:8080`
`histograma` , `ventana`	el histograma de latencia y la ventana del rate	`classifier_request_latency_seconds, 5m`
λ	requests por segundo	`500 en el pico de diseño (≈ 100 por pod con 5 pods)`
t_{CPU}	CPU por request: decode, resize, normalizar	`≈ 6 ms en promedio`
c_{request}	CPU que pide cada pod (requests.cpu)	`1`
u_{HPA}	utilización objetivo del HPA (el autoscaler de pods por CPU)	`0.6`
ρ_{loop}	fracción del tiempo en que el event loop está ocupado con trabajo bloqueante	`≈ 0.8 en el laboratorio, con el decode adentro`
$\bar{t}_{\text{bloqueo}}$	trabajo bloqueante promedio por request dentro del loop	`≈ 20 ms en el laboratorio (0.95 × 12 + 0.05 × 175)`
$t_{\text{keep-alive}}$, t_{idle}	cuánto espera cada lado (uvicorn y el ALB, el balanceador de AWS) antes de cerrar una conexión ociosa	`75 s en uvicorn, 60 s en el ALB`

histogram_quantile(0.99, sum by (le)
(rate(*histograma* **_bucket[** *ventana* **])))**

El **p99** (la latencia bajo la cual queda el 99 % de los requests) de todo el servicio, en PromQL, el lenguaje de consultas de Prometheus: primero se suman los buckets (los conteos por rango de latencia) de todos los pods y después se calcula el cuantil. Nunca promedies el p99 de cada pod.

FÓRMULAS

$$\text{cores} = \lambda \cdot t_{\text{CPU}} = 500\ \frac{\text{req}}{\text{s}} \times 6\ \text{ms} = 3$$

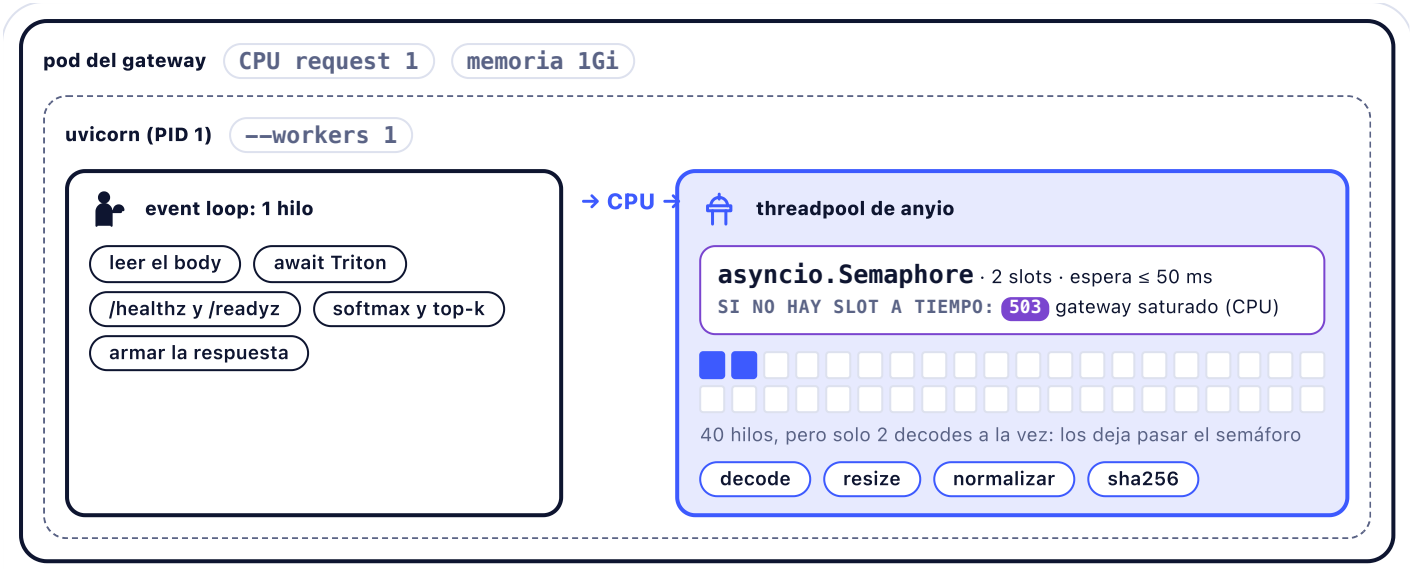
$$\text{pods} = \left\lceil \frac{\lambda \cdot t_{\text{CPU}}}{c_{\text{request}} \cdot u_{\text{HPA}}} \right\rceil = \left\lceil \frac{3}{1 \times 0.6} \right\rceil = 5$$

$$\rho_{\text{loop}} = \lambda \cdot \bar{t}_{\text{bloqueo}} = 40\ \frac{\text{req}}{\text{s}} \times 20\ \text{ms} \approx 0.8$$

$$t^{\text{uvicorn}}_{\text{keep-alive}} = 75\ \text{s} > t^{\text{ALB}}_{\text{idle}} = 60\ \text{s}$$

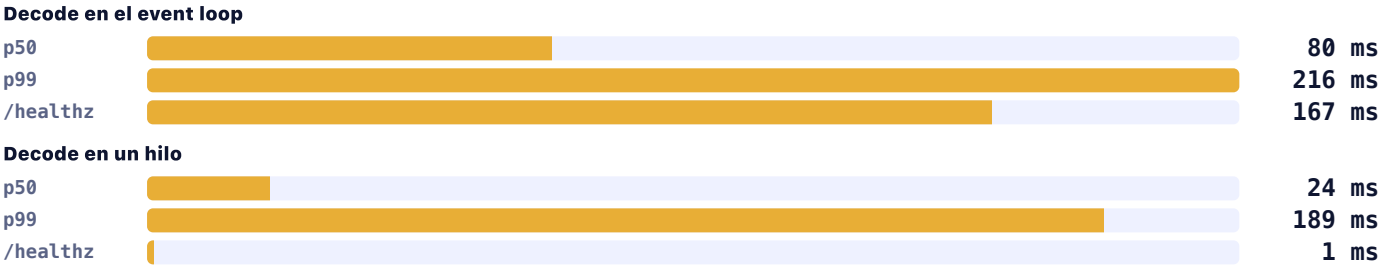
LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
--workers N	N procesos con todo duplicado: memoria, conexiones, métricas. En Kubernetes, 1 por pod y más pods con el HPA.	**--workers 1 en C01**
WEB_CONCURRENCY	Variable de entorno que uvicorn usa como --workers si no pasas el flag.	**si una imagen base la define, tienes workers sin saberlo**
--timeout-keep-alive	Segundos que una conexión ociosa queda abierta (default 5). Debe ser mayor que el idle timeout del balanceador.	**75 > 60 del ALB**
--limit-concurrency	Tope por proceso de conexiones abiertas o tareas; al pasarlo, 503 inmediato en texto plano y sin Retry-After. Cuenta también las conexiones keep-alive ociosas del ALB.	**C01 usa su propio MAX_INFLIGHT**
--backlog	Conexiones que esperan a que el proceso las acepte (default 2048).	**el default**
uvloop · httptools	Versiones más rápidas del event loop y del parser HTTP, escritas en C y Cython; uvicorn las usa solo si están instaladas (--loop auto, --http auto).	**uvloop==0.22.1, httptools==0.8.0**
threadpool de anyio	Los hilos que comparten run_in_threadpool, los endpoints def y las dependencias def: 40 por proceso por defecto.	**C01 deja pasar 2 decodes con un semáforo**
asyncio.to_thread	Corre una función en otro pool: el executor por defecto de asyncio, de min(32, CPUs + 4) hilos.	**el laboratorio de abajo**
GIL	El candado de CPython: un solo hilo ejecuta bytecode de Python a la vez. Pillow y NumPy lo sueltan en el trabajo pesado; un bucle en Python puro, no.	**medido: decode 3.4–3.7× con 4 hilos; Python puro 1.0×**
asyncio.Semaphore	Contador de turnos: limita cuántos decodes corren a la vez, y con ellos la RAM (≈ 320 MB por PNG de 40 MP).	**PREPROCESS_CONCURRENCY=2**
py-spy	Profiler por muestreo: ve el stack del proceso sin modificarlo. top, dump y record.	**py-spy dump --pid 1**
PYTHONASYNCIODEBUG=1	Modo debug de asyncio: avisa de los pasos (callbacks o tareas) que ocupan el loop más de 100 ms.	**Executing <Task …> took 0.182 seconds**
histograma de latencia	Conteos por bucket que se suman entre pods; de ahí sale el p99. Buckets densos alrededor del SLO (la meta de latencia: p99 ≤ 200 ms en C01).	**0.15, 0.2 y 0.3 s en C01**
CPU throttling	Con limits.cpu, el kernel frena al contenedor que gastó su cuota, en ventanas de 100 ms: picos de p99 con CPU promedio baja.	**C01 no pone límite de CPU**



MÁS CAPACIDAD ✓ más pods: HPA de 3 a 12, al 60 % de CPU ✕ --workers N en el mismo pod

LABORATORIO: 40 REQ/S, 1 DE CADA 20 FOTOS DE 12 MP



A escala, de 0 a 216 ms.

lo hace el event loop: esperas de I/O y pasos cortos · CPU pesada: va a un hilo · sin slot a tiempo: 503

Un pod del C01: un proceso uvicorn con un event loop y un threadpool. El loop atiende el I/O; el decode va a los hilos, de a 2. /metrics es un def: también corre en un hilo, sin semáforo. Para más capacidad, más pods, no más workers. Abajo, lo medido en el laboratorio.

EJEMPLO REAL · C01: CÓMO ARRANCA UVICORN EN EL CONTENEDOR, Y POR QUÉ CADA FLAG

DOCKERFILE reference/c01/gateway/Dockerfile (líneas 44–52)

```
# 1 proceso por contenedor (--workers 1): en k8s escalas con réplicas (HPA), cada pod con su
# request de CPU, su métrica de /metrics y su probe. Varios workers en un pod esconden fallas,
# duplican memoria y rompen prometheus_client sin modo multiproceso.
#   --timeout-keep-alive 75   mayor que el idle timeout del ALB (60 s): si el backend cierra primero,
#                             el ALB reusa una conexión muerta y el cliente ve 502 esporádicos
#   --no-access-log           el gateway ya escribe una línea JSON por request; el access log duplicaría
#   --timeout-graceful-shutdown 20  en SIGTERM termina los requests en curso (< terminationGracePeriodSeconds)
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080", "--workers", "1", \
     "--timeout-keep-alive", "75", "--no-access-log", "--timeout-graceful-shutdown", "20"]
```

- `--workers 1` : un proceso por pod. El HPA (el autoscaler que agrega o quita pods según la CPU) ve la CPU real de ese proceso, `/metrics` describe a ese proceso y la probe habla con él. Con 4 workers en un pod, cada scrape de Prometheus cae en un worker distinto y los contadores suben y bajan (sin el modo multiproceso de prometheus_client).
- ¿Y si un proceso no alcanza? Primero, que el event loop no haga CPU: el decode va al threadpool y Pillow suelta el GIL, así que un proceso usa más de un core. Después, más pods: a 500 req/s son ≈ 3 cores de decode, ≈ 5 pods de 1 CPU al 60 % (fórmula de arriba).
- `--timeout-keep-alive 75` : una conexión **keep-alive** queda abierta entre requests para reusarla, y uvicorn la cierra si pasa 5 s ociosa (su default). Si el gateway la cierra justo cuando el ALB la reusa, el ALB responde **502**. Con 75 > 60, siempre cierra primero el ALB (su `idle_timeout.timeout_seconds=60` está en `k8s/gateway.yaml`).
- `--no-access-log` : menos líneas y menos CPU; el registro de cada request es la línea JSON del gateway. El costo: lo que FastAPI responde antes de llegar a una función no deja línea. El C01 cubre el 422 con un exception handler; un 405 o un 404 no dejan rastro en el gateway.

- `--timeout-graceful-shutdown 20`: tras el SIGTERM (la señal con que Kubernetes le pide al proceso que termine), uvicorn espera hasta 20 s los requests en curso y cancela lo que quede (esos terminan en 500). Cabe en los 45 s de gracia: 15 s de preStop (la espera antes del SIGTERM) + 20 s de drenado + 10 s de margen.
- Lista JSON (forma exec): uvicorn es el PID 1 y le llega el SIGTERM directo. Con forma shell, el PID 1 sería `sh` y el apagado ordenado no ocurriría (detalle en la hoja de Docker).

CÓMO RECORDARLA

el loop para esperar, los hilos para calcular, los pods para crecer. Mide antes de mover algo (histograma por tramo, py-spy) y la perilla sale sola: el decode al threadpool, el keep-alive mayor que el del ALB, y más pods antes que más workers.

TRAMPA

subir `--workers` o los hilos para bajar un p99 alto. Si el event loop está bloqueado, más hilos no lo destraban; si el código es Python puro, el GIL no deja usar más cores; y si el p99 son las fotos de 12 MP, cada una tarda lo mismo en cualquier hilo. En el laboratorio, sacar el decode del loop baja el p50 de ≈ 80 a 24 ms, pero el p99 casi no se mueve: para ese, draft mode (el decoder JPEG entrega directo una versión reducida) y fotos más chicas.

En el curso: C01 · Serving · M3 · Little's law: la cuenta que ordena todo · M3 · p50, p95, p99: por qué el promedio miente · M3 · Utilización y colas: por qué no se planea al 100 % · M7 · FastAPI + PyTorch: el punto de partida · M6 · HPA: por qué escalar por CPU no sirve para un servicio de GPU

12 Diagnóstico: el método y la tabla de síntomas

Tu gateway está lento, devuelve 503 o se traba: ¿qué miras primero, qué después, y cómo pasas del síntoma a la causa?

SINTAXIS CON VARIABLES

```
kubectl logs deploy/deploy -n ns --all-pods=true --since=ventana | grep -o '"status": [0-9]*' | sort | uniq -c
kubectl logs deploy/deploy -n ns --all-pods=true --since=ventana | grep -o '"error": "[^"]*"' | sort | uniq -c
```

Código y detalle: el número solo no basta. En C01 hay cuatro 503 distintos y dos 504 distintos, y el campo `error` de la línea JSON dice cuál es. `--all-pods=true` junta los logs de todos los pods del Deployment; cada línea llega con `[pod/NOMBRE/gateway]` delante.

```
histogram_quantile(0.99, sum by (le) (rate(classifier_preprocess_seconds_bucket[5m])))
histogram_quantile(0.99, sum by (le) (rate(classifier_inference_seconds_bucket[5m])))
```

Tramo (cada parte del camino del request), en PromQL, las consultas de Prometheus: ¿creció el preproceso, que es CPU del gateway, o la llamada a Triton? Si es Triton, sus métricas separan la espera en cola del cómputo. Los timeouts también entran a `classifier_inference_seconds` (el C01 mide la llamada en un `finally`): un 504 cae entre 0.15 y 0.3 s, así que el p99 de inferencia sube justo cuando Triton está lento.

```
histogram_quantile(0.99, sum by (le, pod) (rate(classifier_request_latency_seconds_bucket[5m])))
kubectl rollout history deploy/deploy -n ns
```

Alcance: ¿todos los pods o uno? Uno solo apunta a su nodo (un vecino ruidoso: otro pod que le roba CPU); todos apuntan al código, a Triton o a la carga. Y ¿coincide con un deploy? Entonces el **rollback** (volver a la versión anterior) va primero.

```
kubectl logs deploy/deploy -n ns --all-pods=true --since=ventana | grep request_id
py-spy dump --pid pid
```

Un caso concreto: el `X-Request-ID` que devolvió la respuesta lleva a su línea de log, con `preprocess_ms`, `inference_ms` y el error. Y si el proceso parece trabado, `py-spy dump` muestra qué hace cada hilo en este momento.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
deploy, *ns*	el Deployment y su namespace	`gateway, classifier`
ventana	cuánto hacia atrás mirar	`10m`
request_id	el X-Request-ID de la respuesta	`un hex de 32 caracteres`
pid	el proceso de uvicorn	`1 dentro del contenedor`
5xx, 4xx, total	requests por familia de código en classifier_requests_total; page = la alerta despierta a quien está de guardia	`la alerta ClassifierHigh5xxRate de C01`
L	requests en vuelo en un pod	`16 = MAX_INFLIGHT: el pod empieza a rechazar`
λ	requests por segundo que recibe ese pod	`100`
W	tiempo promedio de cada request, por la ley de Little (en vuelo = tasa × tiempo) despejada	`160 ms: algo se frenó`
t_{red}	serializar el tensor (602 KB) y la ida y vuelta por la red hasta el Triton de la zona	`2–4 ms`
t_{cola}, $t_{\text{cómputo}}$	lo que mide Triton: espera en su cola y trabajo en la GPU	`≤ 2 ms y 3–6 ms en el p50`
$t_{\text{espera del loop}}$	lo que tarda el event loop del gateway en volver a atender la respuesta	`≈ 0 si nada lo bloquea`

FÓRMULAS

$$\text{tasa}_{5xx} = \frac{5xx}{\text{total} - 4xx} > 1\% \text{ durante } 5 \text{ min} \Rightarrow \text{page}$$

$$W = \frac{L}{\lambda} = \frac{16}{100 \text{ req/s}} = 160 \text{ ms}$$

$$t_{\text{gateway}\to\text{Triton}} \approx t_{\text{red}} + t_{\text{cola}} + t_{\text{cómputo}} + t_{\text{espera del loop}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
503 gateway saturado: reintenta más tarde	El pod ya tenía 16 requests en vuelo (MAX_INFLIGHT) y rechazó sin leer el body.	`kubectl get hpa (el autoscaler de pods) y el p99`
503 gateway saturado (CPU)	Ningún turno de decode libre en 50 ms: CPU del pod saturada.	`classifier_preprocess_seconds y classifier_image_bytes`

PIEZA	QUÉ HACE	EJEMPLO
503 Triton no disponible	Falla de red hacia Triton: conexión rechazada o reseteada, o DNS.	**pods de Triton y sus endpoints**
503 Triton rechazó la inferencia: …	Triton respondió un error: modelo no cargado, o su cola lo rechazó (Request timeout expired).	**el texto después de los dos puntos**
503 el modelo no está listo en Triton	Lo devuelve /readyz: el pod sale del ALB hasta que Triton esté listo.	**kubectl events --types=Warning**
504 timeout de inferencia	Triton no respondió en min(150 ms, lo que quedaba del presupuesto).	**cola contra cómputo en Triton**
504 se agotó el presupuesto del request	Los 250 ms se gastaron antes de llegar a Triton: body lento o decode lento.	**image_bytes y preprocess_ms de esos requests**
413 · 415 · 400	Rechazos del cliente: más de 10 MB o de 40 MP; tipo no permitido; imagen rota o vacía.	**el campo error de la línea JSON**
405 sin línea de log	FastAPI lo respondió antes de cualquier función (un GET a /v1/classify): no hay log ni métrica del gateway. El 422 sí deja línea en el C01, por su exception handler.	**HTTPCode_Target_4XX_Count del ALB (el balanceador de AWS)**
500 sin campo error	Una excepción que no es HTTPException: un bug, o un cliente que se fue a mitad del upload.	**Exception in ASGI application en kubectl logs**
502 Bad Gateway (HTML del ALB)	El ALB no pudo usar la conexión: el pod la cerró con un request pendiente, por un keep-alive más corto que el del ALB o por apagarse sin drenar.	**--timeout-keep-alive y el apagado**
503 Service Temporarily Unavailable (HTML del ALB)	El ALB no tiene a quién mandar el request: todos los gateways quedaron NotReady (no listos), casi siempre porque Triton cayó.	**kubectl get endpointslices -n classifier**
460 en los access logs del ALB	El cliente cerró la conexión antes de la respuesta (nginx lo anota como 499): su timeout es más corto que tu latencia.	**latency_ms > 300 en los logs**

1 Síntoma con números → 2 Código + detalle → 3 Tramo → 4 Alcance → 5 Un request →
6 El proceso por dentro

1 Síntoma con números

¿Qué código o qué percentil, cuánto y desde cuándo?

MIRA:

classifier_requests_total por status

el p99 del histograma

2 Código + detalle

¿Cuál de los 503 o de los 504?

MIRA:

el campo error de la línea JSON

3 Tramo

¿Preproceso (CPU) o Triton (red, cola, cómputo)?

MIRA:

classifier_preprocess_seconds contra classifier_inference_seconds

4 Alcance

¿Todos los pods o uno? ¿Coincide con un deploy?

MIRA:

by (pod)

`kubectl rollout history`

5 **Un request**

¿Qué le pasó a este request_id?

MIRA:

`kubectl logs … | grep ID`

6 **El proceso por dentro**

¿Qué hace el event loop ahora mismo?

MIRA:

`py-spy dump`

`PYTHONASYNCIODEBUG=1`

> **Mitigar primero, entender después.**

El método, de izquierda a derecha. Cada paso descarta una familia de causas; mitiga apenas sepas lo suficiente (rollback, escalar, sacar un pod) y entiende después.

EJEMPLO REAL · C01: EL RUNBOOK DE CLASSIFIERHIGHP99LATENCY, PASO POR PASO

TEXTO **Pasos 1 a 4 del runbook de ClassifierHighP99Latency (M12, runbooks) como consultas con los nombres de reference/c01 (gateway, monitoring/alerts.yml), más un paso 5 de alcance que agrega esta hoja; sin salidas**

```
# Runbook de ClassifierHighP99Latency (p99 > 200 ms durante 10 min): mitigar primero, entender después
# 1) ¿Qué tramo subió? CPU del gateway o la llamada a Triton (PromQL)
histogram_quantile(0.99, sum by (le) (rate(classifier_preprocess_seconds_bucket[5m])))
histogram_quantile(0.99, sum by (le) (rate(classifier_inference_seconds_bucket[5m])))
# Si es Triton: ¿espera en la cola o cómputo? (ms promedio por request)
sum(rate(nv_inference_queue_duration_us{model="classifier"}[5m]))
  / sum(rate(nv_inference_request_success{model="classifier"}[5m])) / 1000

# 2) ¿Hubo deploy? Si coincide, rollback
kubectl rollout history deploy/gateway -n classifier
kubectl rollout history deploy/triton -n classifier
kubectl rollout undo deploy/gateway -n classifier

# 3) ¿KEDA llegó a su máximo (4 Triton)? Falta GPU
kubectl get hpa -n classifier

# 4) ¿Un solo caller manda de más? Su cuota, no más pods
sum by (source) (rate(classifier_requests_total[5m]))

# 5) ¿Todos los pods o uno?
histogram_quantile(0.99, sum by (le, pod) (rate(classifier_request_latency_seconds_bucket[5m])))
```

- El orden sale del **runbook** de C01 en M12 (la receta escrita de qué hacer ante una alerta): tramo, deploy, capacidad y caller, sus pasos 1 a 4. El paso 5 (¿todos los pods o uno?) lo agrega esta hoja. Cada paso descarta una familia de causas con un vistazo.
- Los dos histogramas del gateway parten el problema en dos: `classifier_preprocess_seconds` es CPU del gateway; `classifier_inference_seconds` es red + cola + cómputo de Triton, más la espera del event loop si está bloqueado.
- Si el p99 subió justo después de un deploy, `rollout undo` va antes que entender: es reversible y corta el impacto. La causa la buscas después, con calma.
- `sum by (source)` separa a los callers (mobile, web, internal y other): si creció uno solo, la respuesta es su cuota, no más pods.
- `by (le, pod)` : si un solo pod tiene el p99 alto, sospecha de su nodo (un vecino ruidoso) antes que del código.
- Lo que este runbook no ve: un 405 (FastAPI lo responde sin pasar por ninguna función) y lo que responde el ALB por su cuenta. Los 504 sí están: entran al histograma de inferencia con lo que esperaron. Para lo demás está la tabla de síntomas de arriba.

CÓMO RECORDARLA

código → detalle → tramo → alcance → request → proceso. El código dice qué familia; el campo `error` , cuál; los histogramas por tramo, dónde; `by (pod)` y el historial de deploys, a quién; el `request_id` , un caso concreto; py-spy, qué hace el proceso ahora.

TRAMPA

diagnosticar con un solo número, o con el promedio. “503” en C01 son cuatro causas distintas, y “504” son dos: Triton no contestó a tiempo (la línea trae `inference_ms` ≈ 150) o el presupuesto se agotó antes de llamarlo (sin `inference_ms`). Y no escales a ciegas: más pods no arreglan un event loop bloqueado ni un Triton lento.

En el curso: C01 · Modos de falla y degradación · C01 · Monitoreo, alertas y reentrenamiento · M12 · Dashboards, runbooks y on-call · M12 · Prometheus, Grafana y alertas por SLO · M3 · Timeouts, reintentos y hedging

Docker y kubectl · Depurar pods: síntoma → comandos → causa típica

13 Errores que debes reconocer rápido

Lees el mensaje: ¿quién lo dijo (tu app, uvicorn, el cliente HTTP, asyncio, Kubernetes o el ALB), qué significa y qué corres primero?

SINTAXIS CON VARIABLES

{"detail":[{"type":" *tipo* **","loc":["** *lugar* **","** *campo* **"],"msg":"** *mensaje* **","input":** *valor* **,"ctx":{"** *limite* **":** *n* **}}]}**

Un **422** de FastAPI: la validación rechazó el request y tu función ni corrió. `loc` dice dónde (query, path, header o body) y qué campo; `input`, lo que llegó; `ctx`, el límite que no se cumplió.

{"detail":" *mensaje* **"}**

Un `HTTPException`: lo decidió tu código (o FastAPI, como `Not Found`). En C01, el mensaje dice cuál de los 503 o de los 504 es.

N **validation error for** *Modelo*
campo
mensaje **[type=** *tipo* **, input_value=** *valor* **, input_type=** *tipo_py* **]**

Un `ValidationError` de Pydantic dentro de tu código, no en el request: el cliente recibe un 500, no un 422.

NIVEL **:** *mensaje*

Una línea de uvicorn (INFO, WARNING o ERROR): arranque, apagado, el puerto, `Exception in ASGI application`.

modulo **.** *Excepcion* **:** *mensaje*

La última línea de un **traceback** (la lista de llamadas que llevó al error): la excepción y su texto. Así hablan aiohttp, tritonclient, httpx y asyncio; las líneas de arriba dicen en qué línea de tu código pasó.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
tipo	el código del error de Pydantic	`greater_than_equal, missing, int_parsing`
lugar	dónde venía el dato	`query, path, header, body`
campo	el campo con problema	`top_k`
mensaje	el texto del error	`Input should be greater than or equal to 1`
valor	lo que llegó; en query y headers, siempre texto	`"0"`
limite, *n*	la regla que no se cumplió	`ge, 1`
N, *Modelo*	cuántos errores (desde 2 dice errors) y en qué modelo de Pydantic	`1, Prediccion`
tipo_py	el tipo de Python de lo que llegó	`str`
NIVEL	la gravedad del log de uvicorn	`INFO, WARNING, ERROR`
modulo . *Excepcion*	quién la lanzó	`tritonclient.utils.InferenceServerException, httpx.ReadTimeout`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
422 {"detail":[…]}	FastAPI validó el request y lo rechazó: tu función no corrió.	`top_k=0`
{"detail":"…"}	Un HTTPException: tu código eligió el código y el texto.	`gateway saturado (CPU)`
Internal Server Error (texto plano)	Una excepción que nadie atrapó; el detalle está en el log, no en la respuesta.	`Exception in ASGI application`
… validation error for …	Pydantic rechazó datos al crear un modelo dentro de tu código: es un 500, no un 422. Si falla el response_model, FastAPI lo dice con otro texto: ResponseValidationError, también 500.	`[type=float_parsing, …]`
ERROR: · WARNING: · INFO:	uvicorn hablando: arranque, apagado, el puerto, requests inválidos.	`Error loading ASGI app.`
aiohttp.ClientError	La familia de errores de red de aiohttp, que usa tritonclient.http.aio; C01 la traduce a 503.	`ClientConnectorError`
InferenceServerException: [código] mensaje	Triton respondió un error; entre corchetes, el código HTTP que mandó Triton: 400 es tu request, 404 un modelo que no conoce, 503 capacidad o estado.	`[404] Request for unknown model…`
httpx.TimeoutException	Si tu gateway usa httpx (C01 no): venció uno de sus cuatro timeouts (connect, read, write, pool), 5 s cada uno por defecto.	`ReadTimeout, PoolTimeout`
httpx.NetworkError · ProtocolError	Si usas httpx: no hubo conexión (ConnectError) o el servidor cortó o violó el protocolo (RemoteProtocolError).	`Server disconnected without sending a response.`

PIEZA	QUÉ HACE	EJEMPLO
RuntimeWarning · RuntimeError de asyncio	Uso incorrecto del event loop: una corrutina sin await, un loop cerrado o ajeno.	`Event loop is closed`
Readiness probe failed	El kubelet (el agente de Kubernetes en cada nodo) no pudo confirmar que el pod está listo (readiness): el pod sale del ALB, el balanceador de AWS.	`HTTP probe failed with statuscode: 503`
502 · 503 · 504 en HTML	Lo respondió el ALB, no tu app: no aparece en tus logs.	`502 Bad Gateway`
FAILED … AssertionError	pytest: un assert no se cumplió; abajo muestra los valores comparados.	`assert 200 == 504`

LOS 26 MENSAJES

`Error loading ASGI app. Could not import module "app.main".`

uvicorn encontró el paquete app, pero no el módulo app.main: un typo (app.mian:app), falta app/main.py, o Python encontró otra carpeta app/ antes que la tuya. Si arrancas desde una carpeta sin app/, el error es otro: ModuleNotFoundError: No module named 'app'. Y si el módulo existe pero le falta una dependencia, ves el ModuleNotFoundError de esa dependencia (No module named 'numpy'). C01 se arranca desde gateway/, donde está el paquete app/.

PRIMERO: `python -c "import app; print(app.__path__)"` (en la carpeta donde corres uvicorn: ¿qué app/ encontró y tiene main.py?)

→ Sección 2 · FastAPI básico: app, rutas, modelos, response_model, códigos, dependencias y lifespan

`AttributeError: 'State' object has no attribute 'inflight'`

El lifespan no corrió, así que app.state no tiene lo que se crea al arrancar (inflight, cpu_slots, triton). Pasa si creas FastAPI() sin lifespan=lifespan, o en un test con TestClient(app) sin with. El request termina en 500.

PRIMERO: `grep -n "FastAPI(" app/main.py`

→ Sección 2 · FastAPI básico: app, rutas, modelos, response_model, códigos, dependencias y lifespan

`RuntimeWarning: coroutine 'infer' was never awaited`

Llamaste a una función async sin await: se creó la corrutina, pero nunca corrió. No es un error: el request sigue sin esa parte (sin inferencia, sin log, sin métrica). Python avisa cuando la corrutina se descarta; con PYTHONASYNCIODEBUG=1 dice además dónde se creó.

PRIMERO: `grep -n "infer(" app/*.py`

→ Sección 3 · async contra hilos: event loop, async def contra def, trabajo bloqueante, threadpool, workers y GIL

`RuntimeError: Event loop is closed`

Algo quiso usar un event loop que ya terminó: un cliente async (aiohttp, httpx) u otro objeto de asyncio creado en un loop y usado en otro. Pasa al crearlos al importar el módulo, o entre tests (TestClient sin with usa un loop por request). Por eso C01 crea el cliente de Triton en el lifespan.

PRIMERO:

```
grep -nE
"ClientSession|AsyncClient|InferenceServerClient"
app/*.py
```

→ Sección 3 · async contra hilos: event loop, async def contra def, trabajo bloqueante, threadpool, workers y GIL

`{"detail":[{"type":"greater_than_equal","loc":["query","top_k"],"msg":"Input should be greater than or equal to 1","input":"0","ctx":{"ge":1}}]}`

Un 422 de FastAPI: el request no cumplió el contrato y tu función ni corrió. type es el código del error; loc, dónde (query) y qué campo; input, lo que llegó (en la query siempre es texto); ctx, el límite. FastAPI quita el campo url que agrega Pydantic.

PRIMERO:

```
curl -s -X POST "localhost:8080/v1/classify?top_k=0" -H
"Content-Type: image/jpeg" --data-binary @foto.jpg
```

→ Sección 4 · Validación y contrato: Pydantic v2, uploads con límite, 400/413/415/422, versionado y OpenAPI

`ValidationError: 1 validation error for Prediccion · score · Input should be a valid number, unable to parse string as a number [type=float_parsing, input_value='alto', input_type=str]`

Pydantic rechazó datos dentro de tu código (al armar un modelo con la respuesta de otro servicio, o con la config), no en el request: el cliente recibe un 500, no un 422. Debajo viene un enlace a errors.pydantic.dev con la versión (2.13 en C01). En Pydantic v1 el mismo error decía value is not a valid float.

PRIMERO:

```
kubectl logs deploy/gateway -n classifier --all-
pods=true --since=10m | grep -B 5 "validation error for"
```

→ Sección 4 · Validación y contrato: Pydantic v2, uploads con límite, 400/413/415/422, versionado y OpenAPI

`Cannot connect to host triton.classifier.svc.cluster.local:8000 · httpx.ConnectError: All connection attempts failed`

No hubo conexión con el servidor de abajo: nada escucha en ese host y puerto (conexión rechazada) o el DNS no resuelve el nombre. Lo primero es el ClientConnectorError de aiohttp, que usa tritonclient.http.aio: C01 lo traduce a 503 Triton no disponible con Retry-After: 1. Lo segundo es lo que verías si tu gateway usara httpx async (con el cliente síncrono: [Errno 111] Connection refused). Si una NetworkPolicy descarta los paquetes, no ves ninguno de los dos: la conexión se cuelga y, en C01, vence el timeout (504).

PRIMERO:

```
kubectl get endpointslices -n classifier -l
kubernetes.io/service-name=triton
```

→ Sección 5 · Llamar al servidor de modelos: cliente, pool, timeouts, reintentos y deadline

`httpx.RemoteProtocolError: Server disconnected without sending a response.`

Si usas httpx (C01 no): el servidor cerró la conexión sin responder. Lo típico: reusaste del pool una conexión keep-alive que el servidor acababa de cerrar por inactividad (uvicorn cierra a los 5 s por defecto, y httpx guarda las ociosas 5 s: empatan). La regla es la del ALB: el cliente suelta la conexión antes que el servidor (keepalive_expiry menor). También pasa si el servidor se reinició a mitad del request.

PRIMERO:

```
grep -nE "AsyncClient|Limits|keepalive_expiry" app/*.py
```

(¿keepalive_expiry menor que el keep-alive del servidor de abajo?)

→ Sección 5 · Llamar al servidor de modelos: cliente, pool, timeouts, reintentos y deadline

`tritonclient.utils.InferenceServerException: [404] Request for unknown model: 'classifier' is not found`

Triton no conoce un modelo con ese nombre: un MODEL_NAME mal escrito, o un modelo que nunca se cargó. Entre corchetes va el código HTTP que mandó Triton: 404 en la 24.08 del C01 (hasta la 23.12, 400); .message() da el texto sin él. En C01 le llega al cliente como 503 Triton rechazó la inferencia: … con Retry-After: 1.

PRIMERO:

```
curl -s -o /dev/null -w '%{http_code}\n'
localhost:8000/v2/models/classifier/ready
```

(con kubectl port-forward -n classifier svc/triton 8000:8000)

→ Sección 5 · Llamar al servidor de modelos: cliente, pool, timeouts, reintentos y deadline

`httpx.ReadTimeout`

Si usas httpx: esperó datos del servidor más que el timeout de lectura (5 s por defecto). Con el cliente async el texto suele venir vacío; con el síncrono, timed out. Los timeouts de httpx son por operación (connect, read, write, pool), no un total: para un presupuesto por request, envuelve la llamada con asyncio.timeout o asyncio.wait_for, como C01 con tritonclient.

PRIMERO: `grep -nE "Timeout|timeout=" app/*.py`

→ Sección 5 · Llamar al servidor de modelos: cliente, pool, timeouts, reintentos y deadline

`httpx.PoolTimeout`

Si usas httpx: ninguna conexión del pool se liberó antes del timeout de pool (5 s por defecto). Tienes más requests en vuelo hacia ese servidor que max_connections (100 por defecto), y cada uno tarda. Casi nunca se arregla agrandando el pool: el servidor de abajo está lento, le mandas de más, o hay respuestas en streaming sin cerrar que retienen conexiones.

PRIMERO:

```
histogram_quantile(0.99, sum by (le)
(rate(classifier_inference_seconds_bucket[5m])))
```

para ver la latencia hacia el servidor de abajo, en C01 Triton

→ Sección 5 · Llamar al servidor de modelos: cliente, pool, timeouts, reintentos y deadline

`503 {"detail":"gateway saturado: reintenta más tarde"} · Retry-After: 1`

Load shedding del C01: el pod ya tenía 16 requests en vuelo (MAX_INFLIGHT) y rechazó este en microsegundos, sin leer el body. Retry-After dice en cuántos segundos reintentar. Si se sostiene, algo se frenó (16 en vuelo a 100 req/s son 160 ms por request) o falta capacidad.

PRIMERO: `kubectl get hpa gateway -n classifier`

→ Sección 6 · Backpressure y control de admisión: límite en vuelo, 429 contra 503, load shedding

`Triton rechazó la inferencia: Request timeout expired`

La cola de Triton rechazó el request porque esperó más de 100 ms (default_queue_policy con REJECT): la GPU no da abasto. tritonclient lo muestra como [503] Request timeout expired, y C01 lo devuelve como 503 con Retry-After: 1.

PRIMERO:

```
sum(rate(nv_inference_queue_duration_us{model="classifie
r"}[5m])) /
sum(rate(nv_inference_request_success{model="classifier"
}[5m])) / 1000
```

→ Sección 6 · Backpressure y control de admisión: límite en vuelo, 429 contra 503, load shedding

`ValueError: Out of range float values are not JSON compliant: nan`

Quisiste devolver NaN o infinito en un JSON. En una ruta sin response_model, como classify, FastAPI usa el JSONResponse de Starlette, con allow_nan=False: el cliente recibe un 500, y si tu finally ya anotó el request, la línea de log y la métrica dicen 200 (el finally corre antes de serializar). Por eso el C01 revisa los logits con np.isfinite apenas llegan de Triton y responde un 500 propio, que sí se cuenta: en sus logs no vas a ver este mensaje. Con un response_model, Pydantic lo escribe como null, sin error. En un clasificador, el NaN suele venir de logits rotos (un engine mal construido, un FP16 desbordado). Desde Python 3.12 el mensaje termina con el valor (: nan); en 3.11 o antes, sin él.

PRIMERO:

```
kubectl logs deploy/gateway -n classifier --all-
pods=true --since=10m | grep -B 20 "not JSON compliant"
```

· en el C01:

```
kubectl logs deploy/gateway -n classifier --all-
pods=true --since=10m | grep -c 'logits no finitos'
```

→ Sección 7 · Errores y degradación: un código por falla, la escalera de C01 y nada de detalles internos

`Internal Server Error · Exception in ASGI application`

Una excepción que nadie atrapó. El cliente recibe un 500 en texto plano, sin detalle (bien: no filtras nada); el detalle está en el log, donde uvicorn imprime Exception in ASGI application y el traceback. En C01, la línea JSON de ese request queda con status 500 y sin campo error.

PRIMERO:

```
kubectl logs deploy/gateway -n classifier --all-pods=true --since=10m | grep -A 30 "Exception in ASGI application"
```

→ Sección 7 · Errores y degradación: un código por falla, la escalera de C01 y nada de detalles internos

`Readiness probe failed: HTTP probe failed with statuscode: 503`

El /readyz del gateway respondió 503: el modelo no está listo en Triton, o Triton no contestó en 0.5 s. El pod sale del ALB sin reiniciarse. Si pasa en todos los gateways a la vez, el problema es Triton, no el gateway.

PRIMERO: `kubectl events -n classifier --types=Warning`

→ Sección 8 · Health, readiness y apagado ordenado

`Shutting down · Waiting for connections to close. (CTRL+C to force quit)`

uvicorn recibió SIGTERM (o Ctrl+C): cerró el puerto y espera los requests en curso, hasta --timeout-graceful-shutdown (20 s en C01; sin el flag, espera sin límite). Si se acaba el plazo, cancela lo que queda (Cancel N running task(s), timeout graceful shutdown exceeded) y esos requests terminan en 500. Después: Application shutdown complete y Finished server process.

PRIMERO: `kubectl logs -f POD -n classifier`

→ Sección 8 · Health, readiness y apagado ordenado

`TypeError: Object of type float32 is not JSON serializable`

json.dumps no sabe escribir tipos de NumPy: float32 e int64 fallan (float64 pasa, porque es una subclase de float). En C01 la línea de log se escribe en un finally: si el record trae un np.float32, el error reemplaza la respuesta y el request termina en 500. Por eso el código convierte con float() e int(). Si lo devuelves desde una ruta sin response_model (como classify), FastAPI falla con otro texto: un ValueError que incluye 'numpy.float32' object is not iterable.

PRIMERO: `grep -nE "float\(|int\(" app/main.py`

→ Sección 9 · Observabilidad: una línea JSON por request, request ID, métricas y trazas

`prometheus_client.registry.DuplicateTimeseries: Duplicated timeseries in CollectorRegistry: {'classifier_requests_total', …}`

Registraste dos veces una métrica con el mismo nombre en el registro global de prometheus_client: el módulo se importó dos veces con nombres distintos, un test lo recargó, o en un notebook re-ejecutaste la celda que crea el Counter. Desde prometheus-client 0.26.0 (la del C01) la excepción es DuplicateTimeseries, una subclase de ValueError; antes salía ValueError: con el mismo texto. Las métricas se crean una vez, al nivel del módulo.

PRIMERO: `grep -rnE "Counter\(|Histogram\(" app/`

→ Sección 9 · Observabilidad: una línea JSON por request, request ID, métricas y trazas

`TypeError: object MagicMock can't be used in 'await' expression`

Reemplazaste un cliente async con MagicMock: lo que devuelve su método es otro mock, y no se puede esperar con await. Usa AsyncMock o, mejor, un fake con los mismos métodos async y las mismas excepciones, como el FakeTriton del C01. En Python 3.14 el texto cambió a 'MagicMock' object can't be awaited.

PRIMERO: `grep -n "MagicMock" tests/*.py`

→ Sección 10 · Tests: pytest, TestClient, el fake de Triton y las pruebas de carga

`async def functions are not natively supported.`

pytest no sabe correr solo un test async def: desde pytest 8.4 el test falla (antes se saltaba con un aviso). Márcalo con @pytest.mark.anyio (el plugin viene con anyio, que ya instala FastAPI), usa pytest-asyncio, o escribe el test con def y TestClient, como C01.

PRIMERO: `grep -n "async def test_" tests/*.py`

→ Sección 10 · Tests: pytest, TestClient, el fake de Triton y las pruebas de carga

`RuntimeError: The starlette.testclient module requires the httpx2 package to be installed.`

TestClient necesita un cliente HTTP por debajo. Desde Starlette 1.2 busca httpx2 y, si solo encuentra httpx, funciona pero avisa que está deprecado; sin ninguno de los dos, falla con este error. C01 fija httpx2==2.13.1 en requirements-dev.txt.

PRIMERO: `pip install -r requirements-dev.txt`

→ Sección 10 · Tests: pytest, TestClient, el fake de Triton y las pruebas de carga

`Executing <Task pending name='Task-3' coro=<request() running at carga.py:30> … took 0.182 seconds`

El modo debug de asyncio (PYTHONASYNCIODEBUG=1) avisa que una tarea ocupó el event loop 182 ms seguidos: en ese tiempo no avanzó ningún otro request. El umbral es 100 ms (loop.slow_callback_duration). Dice qué corrutina fue y la línea donde quedó pausada después, el siguiente await: el código bloqueante (CPU o I/O sin await) está justo antes de esa línea.

PRIMERO:

```
PYTHONASYNCIODEBUG=1 uvicorn app.main:app --port 8080 --loop asyncio
```

→ Sección 11 · Rendimiento: workers de uvicorn, CPU para el decode, keep-alive, perfilar y medir el p99

`502 Bad Gateway`

Una página HTML del ALB: el pod cerró la conexión (TCP RST o FIN) con un request pendiente. Si son esporádicos, casi siempre es keep-alive: el backend cierra la conexión ociosa antes que el ALB (uvicorn, a los 5 s por defecto; el ALB, a los 60). En los deploys, un pod que se apagó sin drenar. No aparece en tus logs.

PRIMERO: `grep -n "timeout-keep-alive" Dockerfile`

→ Sección 11 · Rendimiento: workers de uvicorn, CPU para el decode, keep-alive, perfilar y medir el p99

[Errno 48] error while attempting to bind on address ('127.0.0.1', 8080): [errno 48] address already in use

Otro proceso ya usa ese puerto: otro uvicorn que quedó corriendo, o docker compose. uvicorn lo imprime como ERROR al arrancar y sale. Es el texto del loop de asyncio en Python 3.13 y 3.14; en 3.11 o antes, y con uvloop (instalado en C01), no repite el [errno …] del medio. Con --workers 2 o más, o con --reload, es más corto: [Errno 48] Address already in use. En Linux el número es 98.

PRIMERO: `lsof -nP -iTCP:8080 -sTCP:LISTEN`

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

starlette.requests.ClientDisconnect

El cliente cerró la conexión mientras el gateway leía el body con request.stream(). La excepción no trae texto. Si nadie la atrapa, el request cuenta como 500 sin campo error y uvicorn imprime Exception in ASGI application, aunque el servidor esté sano. El C01 la atrapa al leer el body y la anota como 499, sin traceback: en sus logs vas a ver la línea JSON, no este mensaje. En el ALB, un cliente que corta aparece como 460 (o 400, si cortó antes de mandar todo el body); en nginx, como 499.

PRIMERO:

```
kubectl logs deploy/gateway -n classifier --all-pods=true --since=1h | grep -c 'requests.ClientDisconnect$'
```

· en el C01:

```
kubectl logs deploy/gateway -n classifier --all-pods=true --since=1h | grep -c '"status": 499'
```

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

EJEMPLO REAL · TRES ERRORES PROVOCADOS CON PYTHON 3.13, LA VERSIÓN DEL CONTENEDOR DEL C01

TEXTO Salidas reales con Python 3.13.15 en macOS; … marca líneas omitidas del traceback

```
$ python3.13 -c 'import asyncio
async def infer(): return [0.1, 0.9]
async def main(): infer()
asyncio.run(main())'
<string>:3: RuntimeWarning: coroutine 'infer' was never awaited
RuntimeWarning: Enable tracemalloc to get the object allocation traceback

$ python3.13 -c 'import json; json.dumps({"score": float("nan")}, allow_nan=False)'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  …
ValueError: Out of range float values are not JSON compliant: nan

$ python3.13 -c 'import asyncio
from unittest.mock import MagicMock
async def classify(client): return await client.infer()
asyncio.run(classify(MagicMock()))'
Traceback (most recent call last):
  …
  File "<string>", line 3, in classify
    async def classify(client): return await client.infer()
                                       ^^^^^^^^^^^^^^^^^^^^
TypeError: object MagicMock can't be used in 'await' expression
```

- Lee de abajo hacia arriba: la última línea es la excepción y su texto; la primera línea con `File` que apunta a tu código dice dónde. Las de en medio suelen ser de librerías.
- Un `RuntimeWarning` no detiene nada: la corrutina sin await simplemente no corre y el programa termina bien. Por eso es peligrosa: no hay 500, solo falta trabajo (una inferencia, un log, una métrica).
- `: nan` al final aparece desde Python 3.12 (también en 3.13 y 3.14); en 3.11 o antes, el mensaje termina en `compliant`. Y el de MagicMock cambió en 3.14 a `'MagicMock' object can't be awaited`: busca por la parte estable del mensaje.
- Quién habla: `{"detail": …}` en JSON es tu app (FastAPI); `Internal Server Error` en texto plano es un 500 que nadie atrapó; `ERROR:` o `INFO:` es uvicorn; `Readiness probe failed` es el kubelet; una página HTML de 502 o 503 es el ALB. En C01 traen `X-Request-ID` las respuestas que arma `classify` y los 422, que pasan por su exception handler; un 500 sin atrapar, no.

CÓMO RECORDARLA

quién habla dice dónde mirar. JSON con `detail` es tu app; `ERROR:` o `INFO:` es uvicorn; `aiohttp`, `InferenceServerException` o `httpx.` es la llamada al servidor de modelos; `RuntimeWarning` y `RuntimeError` de asyncio son el event loop; `Warning … probe failed` es el kubelet; HTML es el ALB.

TRAMPA

buscar el texto exacto sin mirar la versión. Python 3.13 cambió los nombres de los códigos (la línea de estado dice `413 Content Too Large` y `422 Unprocessable Content`; antes, `Request Entity Too Large` y `Unprocessable Entity`), 3.14 cambió el texto de MagicMock, Pydantic v2 cambió todos sus mensajes (`value is not a valid integer` pasó a `Input should be a valid integer…`) y pytest 8.4 convirtió el aviso de los tests async en una falla. Busca por la parte estable: el tipo de excepción y las palabras clave.

En el curso: C01 · Serving · C01 · Modos de falla y degradación · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · M15 · Degradación con gracia: contener la falla

---

APIs, defaults, códigos y mensajes revisados en octubre de 2026 contra la documentación oficial de FastAPI, Starlette, Pydantic, uvicorn, AnyIO, httpx, aiohttp, tritonclient, prometheus_client y OpenTelemetry, contra los RFC de HTTP y contra los archivos de `reference/c01` del curso. Las versiones son las del C01 (Python 3.13, FastAPI 0.141.1, Starlette 1.7.0, Pydantic 2.13.5, uvicorn 0.54.0, tritonclient 2.72.0, aiohttp 3.14.3, prometheus-client 0.26.0, Pillow 12.3.0); cuando la versión actual cambia algo, la sección lo dice. De los 15 laboratorios, 10 se corrieron de verdad en un Mac; los demás se verificaron contra la documentación. Si tu terminal dice otra cosa, créele a tu terminal.

© 2026 Erik Palacios · Contenido bajo CC BY-NC-ND 4.0 · ML desde cero · Reportar un error