PuigFlows

Guía técnica · Proyectos Reales

asyncio en Producción: No Bloquear el Event Loop, TaskGroups, Cancelación y Concurrencia Acotada

Guía práctica de asyncio para servicios Python en producción: por qué bloquear el event loop degrada todo el proceso sin lanzar ni una excepción, cómo detectarlo midiendo el lag del loop y con la introspección de Python 3.14, concurrencia estructurada con TaskGroup y manejo de ExceptionGroup con except*, el bug de la tarea que el recolector de basura hace desaparecer, timeouts con presupuesto propagado entre servicios, cancelación correcta con limpieza y shield, concurrencia acotada con semáforos y colas con backpressure, primitivas de sincronización y qué cambia con eager tasks, python -m asyncio pstree y free-threading. Con código listo para producción y una checklist de despliegue.

Nivel: Intermedio · 8577 palabras · 39 min de lectura

2026-08-16 · puigflows.com

Guía práctica de asyncio para servicios que ya están en producción. El código está pensado para Python 3.12 o superior; cuando algo requiere una versión concreta lo indico de forma explícita.

El modelo mental: un hilo, muchas tareas y cero interrupciones

Un programa asyncio se ejecuta en un solo hilo. El event loop mantiene una cola de tareas listas, ejecuta una hasta que esa tarea cede el control con await, y entonces pasa a la siguiente. No hay preempción: nadie le quita la CPU a nadie. Esa propiedad es exactamente lo que permite atender miles de conexiones lentas con un consumo de memoria ridículo, y es también lo que hace que un solo descuido congele el servicio entero.

De ahí sale la única regla que importa de verdad: nunca bloquees el hilo del event loop. El resto de esta guía son corolarios de esa frase.

1. El pecado original: bloquear el event loop

El fallo más caro de asyncio no es el que revienta con una excepción, sino el que degrada la latencia de todo el proceso sin dejar rastro en los logs. Mira este endpoint, que parece asíncrono:

```
# ANTI-PATRÓN: requests es síncrono. Bloquea el hilo entero.
import requests
from fastapi import FastAPI

app = FastAPI()

@app.get("/perfil/{user_id}")
async def perfil(user_id: str):
```

```
    r = requests.get(f"https://api.interna/users/{user_id}", timeout=5)
    return r.json()
```

La firma dice async def, pero requests.get() se queda esperando en el socket sin ceder el control. Durante esos 300 ms -- o esos 5 segundos si la dependencia va mal -- el event loop no puede atender ninguna otra petición. Con 50 peticiones por segundo, la latencia p99 de endpoints que no tienen nada que ver se dispara y el gráfico no explica por qué.

La versión correcta usa un cliente asíncrono y, sobre todo, lo reutiliza:

```
# CORRECTO: cliente asíncrono con pool de conexiones a nivel de proceso.
import httpx
from contextlib import asynccontextmanager
from fastapi import FastAPI

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Un único cliente para todo el proceso: reutiliza conexiones TCP y TLS.
    app.state.http = httpx.AsyncClient(
        timeout=httpx.Timeout(5.0, connect=2.0),
        limits=httpx.Limits(max_connections=100, max_keepalive_connections=20),
    )
    yield
    await app.state.http.aclose()

app = FastAPI(lifespan=lifespan)

@app.get("/perfil/{user_id}")
async def perfil(user_id: str):
    r = await app.state.http.get(f"https://api.interna/users/{user_id}")
    r.raise_for_status()
    return r.json()
```

Crear un AsyncClient nuevo en cada petición es un error frecuente y silencioso: pierdes el keep-alive, pagas un handshake TLS por llamada y bajo carga acabas agotando puertos efímeros.

El bloqueo que no parece bloqueo

Las llamadas de red síncronas son fáciles de detectar en una revisión de código. Estas no:

- Drivers de base de datos síncronos usados dentro de una corrutina: psycopg2, el ORM de Django en modo síncrono, SQLAlchemy sin su motor asyncio.
- Lectura y escritura de ficheros con open(). El disco parece instantáneo hasta que el volumen de red se degrada.
- Trabajo puro de CPU: parsear un JSON de 40 MB, comprimir, hashear contraseñas con bcrypt o Argon2, generar un PDF.
- time.sleep() en lugar de asyncio.sleep(), normalmente escondido dentro de una utilidad de reintentos heredada.
- Resolución DNS síncrona en librerías que no la delegan al loop.
- Un handler de logging que escribe por red sin buffer.

Cómo detectarlo antes de que lo detecten tus usuarios

El modo debug del loop avisa cuando un callback tarda demasiado. Es la forma más barata de encontrar bloqueos en desarrollo y en staging:

```
import asyncio, logging

async def main():
    loop = asyncio.get_running_loop()
    loop.slow_callback_duration = 0.1   # avisa si algo ocupa el loop >100 ms
    await servir()

logging.basicConfig(level=logging.DEBUG)
asyncio.run(main(), debug=True)

# Salida típica:
# Executing <Task pending name='Task-7' coro=<perfil()>> took 1.402 seconds
```

Para producción, un guardarraíl mucho mejor consiste en medir el retraso del propio loop: programas un despertar cada 250 ms y compruebas cuánto se desvía de lo previsto. Si el loop está sano la desviación es de microsegundos; si alguien lo está bloqueando, se dispara.

```
import asyncio, time
from prometheus_client import Gauge

LAG = Gauge("event_loop_lag_seconds", "Retraso del event loop")

async def monitor_lag(intervalo: float = 0.25) -> None:
    while True:
        inicio = time.perf_counter()
        await asyncio.sleep(intervalo)
        LAG.set((time.perf_counter() - inicio) - intervalo)
```

Alerta cuando esa métrica supere unos 50 ms sostenidos. Es, con diferencia, la señal que más rápido apunta al culpable real cuando la queja es "todo va lento y no sabemos por qué".

Cuando el bloqueo es inevitable: hilos

A veces no hay alternativa asíncrona: un SDK propietario, una librería antigua, una función de CPU. Para eso está asyncio.to_thread(), que ejecuta la función en el executor por defecto y devuelve el control al loop mientras tanto.

```
import asyncio
from argon2 import PasswordHasher

ph = PasswordHasher()

async def verificar_password(hash_guardado: str, password: str) -> bool:
    # Argon2 consume CPU a propósito: sácalo del event loop.
    return await asyncio.to_thread(ph.verify, hash_guardado, password)
```

Dos matices que suelen costar un incidente:

- El executor por defecto es pequeño. El ThreadPoolExecutor por defecto usa min(32, os.cpu_count() + 4) hilos. Si mandas ahí todas tus llamadas bloqueantes, ese número se convierte en tu límite real de concurrencia y las tareas se encolan en silencio. Crea executors dedicados por tipo de trabajo para que un cuello de botella no contamine al resto.
- Los hilos no arreglan la CPU. Con el GIL activado, el trabajo intensivo en CPU se sigue serializando. Para eso está ProcessPoolExecutor, o mejor aún, sacar el cálculo del proceso web.

```
import asyncio
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor

io_pool = ThreadPoolExecutor(max_workers=16, thread_name_prefix="io")
cpu_pool = ProcessPoolExecutor(max_workers=4)
```

```
async def leer_legacy(ruta: str) -> bytes:
    loop = asyncio.get_running_loop()
    return await loop.run_in_executor(io_pool, _leer_sincrono, ruta)

async def render_pdf(datos: dict) -> bytes:
    loop = asyncio.get_running_loop()
    # ProcessPool: argumentos y retorno deben ser serializables con pickle.
    return await loop.run_in_executor(cpu_pool, _render_pesado, datos)
```

Y un detalle poco documentado que muerde: el future que devuelve run_in_executor no se puede cancelar de verdad una vez la función ha empezado. Si cancelas el await, tu corrutina sigue adelante pero el hilo continúa trabajando hasta terminar. Diseña esas funciones para que sean cortas o para que consulten una bandera de parada.

2. TaskGroup: el reemplazo moderno de gather

Durante años, lanzar trabajo concurrente en asyncio significaba asyncio.gather(). Funciona, pero su comportamiento ante errores es traicionero: con la configuración por defecto, si una de las corrutinas falla, gather propaga esa excepción de inmediato y deja el resto corriendo. Esas tareas huérfanas siguen ocupando conexiones y pueden terminar escribiendo en tu base de datos mucho después de que hayas devuelto un 500 al cliente.

asyncio.TaskGroup (Python 3.11+) implementa concurrencia estructurada: el bloque async with no sale hasta que todas las tareas hijas han terminado y, si una falla, las demás se cancelan automáticamente.

```
import asyncio

async def cargar_dashboard(http, user_id: str) -> dict:
    async with asyncio.TaskGroup() as tg:
        t_perfil  = tg.create_task(get_perfil(http, user_id),  name="perfil")
        t_pedidos = tg.create_task(get_pedidos(http, user_id), name="pedidos")
        t_avisos  = tg.create_task(get_avisos(http, user_id),  name="avisos")
    # Al salir del bloque, las tres han terminado con éxito. Garantizado.
    return {
        "perfil":  t_perfil.result(),
        "pedidos": t_pedidos.result(),
        "avisos":  t_avisos.result(),
    }
```

Fíjate en dónde se leen los resultados: después del bloque. Dentro del async with las tareas todavía pueden no haber terminado. El parámetro name= en create_task de un TaskGroup está disponible desde Python 3.14 y merece la pena usarlo, porque aparece en las trazas y en las herramientas de introspección.

Cuando algo falla, TaskGroup lanza un ExceptionGroup, porque pueden haber fallado varias tareas a la vez. Se maneja con except*:

```
try:
    datos = await cargar_dashboard(http, user_id)
except* httpx.TimeoutException as eg:
    # eg.exceptions contiene todos los timeouts que ocurrieron
    log.warning("timeouts en dashboard", n=len(eg.exceptions))
    raise HTTPException(504, "Servicio lento")
except* PermissionError:
    raise HTTPException(403, "Sin acceso")
```

Aquí está la trampa: un except Exception normal no captura un ExceptionGroup por su contenido, solo coincidiría por el tipo del grupo. Es el error clásico al migrar de gather a TaskGroup -- el manejo de

errores existente deja de funcionar y las excepciones suben sin filtrar hasta el borde de la aplicación.

¿Sigue habiendo sitio para gather? Sí, en un caso concreto: cuando quieres los resultados parciales aunque algo falle.

```
# Vista agregada donde un fallo parcial es aceptable.
resultados = await asyncio.gather(
    get_perfil(http, uid),
    get_pedidos(http, uid),
    get_recomendaciones(http, uid),
    return_exceptions=True,
)
perfil, pedidos, recos = [
    None if isinstance(r, BaseException) else r for r in resultados
]
```

Con return_exceptions=True las excepciones vuelven como valores, así que tienes que inspeccionarlas. Si no lo haces, un fallo se convierte en un objeto excepción viajando por tu lógica de negocio hasta provocar un AttributeError incomprensible tres capas más abajo.

3. La tarea que desaparece

Este bug se lleva por delante trabajos en segundo plano y es casi imposible de reproducir en local, porque depende del recolector de basura.

```
# ANTI-PATRÓN: la tarea puede desaparecer a mitad de ejecución.
async def handler(evento):
    asyncio.create_task(enviar_metricas(evento))   # nadie guarda la referencia
    return {"ok": True}
```

El event loop solo mantiene una referencia débil a las tareas. Si nadie más guarda una referencia fuerte, el recolector puede destruir la tarea antes de que termine. En desarrollo funciona siempre; bajo carga, un porcentaje impredecible de tus métricas, correos o webhooks simplemente no se envía. Y no hay error que buscar en los logs.

La solución consiste en guardar la referencia y limpiarla al terminar:

```
_tareas_de_fondo: set[asyncio.Task] = set()

def lanzar_en_segundo_plano(coro, *, nombre: str) -> asyncio.Task:
    tarea = asyncio.create_task(coro, name=nombre)
    _tareas_de_fondo.add(tarea)                      # referencia fuerte
    tarea.add_done_callback(_tareas_de_fondo.discard)
    tarea.add_done_callback(_log_si_falla)
    return tarea

def _log_si_falla(tarea: asyncio.Task) -> None:
    if tarea.cancelled():
        return
    exc = tarea.exception()
    if exc is not None:
        log.error("tarea de fondo fallida",
                  tarea=tarea.get_name(), exc_info=exc)
```

El segundo callback resuelve otro problema real: una tarea que falla sin que nadie llame a result() solo produce un aviso de "exception was never retrieved" cuando el objeto se destruye, a veces minutos después y sin contexto útil. Registrar el error explícitamente convierte un misterio en una línea de log accionable.

Un apunte de diseño: para trabajo de fondo que debe sobrevivir a un reinicio, la respuesta correcta no

es una tarea de asyncio sino una cola persistente. Una tarea en memoria muere con el proceso, y los despliegues son frecuentes.

4. Timeouts: el que falta siempre es el que te tumba

Una corrutina sin timeout puede esperar para siempre. En un servicio con concurrencia acotada, unas cuantas tareas colgadas agotan el pool y la caída se ve como saturación, no como el timeout que faltaba. La causa raíz queda escondida detrás del síntoma.

Desde Python 3.11, asyncio.timeout() es la herramienta preferida porque funciona como gestor de contexto y cubre bloques enteros, no una sola espera:

```
import asyncio

async def obtener_precio(http, sku: str) -> float:
    async with asyncio.timeout(2.0):
        r = await http.get(f"/precios/{sku}")
        datos = r.json()
        return await enriquecer(datos)    # el límite cubre TODO el bloque
```

Al vencer el plazo, asyncio cancela la tarea desde dentro y convierte esa cancelación en un TimeoutError. La diferencia con asyncio.wait_for() es de ergonomía: wait_for envuelve una única espera, mientras que timeout() envuelve una secuencia de operaciones.

En sistemas distribuidos, lo que de verdad quieres no es un timeout por llamada sino una fecha límite que se propaga. Si al usuario le prometes una respuesta en 3 segundos, cada salto aguas abajo debe respetar el tiempo que queda de ese presupuesto en lugar de reiniciar el suyo:

```
import asyncio, contextvars, time

_deadline: contextvars.ContextVar[float | None] = contextvars.ContextVar(
    "deadline", default=None
)

def restante(por_defecto: float) -> float:
    d = _deadline.get()
    return por_defecto if d is None else max(0.0, d - time.monotonic())

async def con_presupuesto(segundos: float, coro):
    _deadline.set(time.monotonic() + segundos)
    async with asyncio.timeout(segundos):
        return await coro

async def llamar_inventario(http, sku: str):
    # Nunca gasta más de lo que le queda al presupuesto global.
    async with asyncio.timeout(min(1.5, restante(1.5))):
        return await http.get(f"/inventario/{sku}")
```

Usa siempre time.monotonic() para plazos. El reloj de pared puede saltar hacia atrás con un ajuste NTP y dejarte con timeouts negativos o, peor, eternos.

Y recuerda la limitación fundamental: asyncio.timeout() no cubre el código bloqueante. Si una llamada síncrona se come el hilo, el loop ni siquiera puede ejecutar el temporizador. El timeout se dispara cuando el bloqueo termina, que es justo cuando ya no sirve de nada.

5. Cancelación: la parte que casi todo el mundo hace mal

La cancelación en asyncio se implementa lanzando asyncio.CancelledError dentro de la corrutina en su

siguiente punto de await. Desde Python 3.8 esa excepción hereda de BaseException y no de Exception, precisamente para que un except Exception descuidado no se la trague.

```
# ANTI-PATRÓN: convierte una tarea cancelable en una que no muere nunca.
async def worker():
    while True:
        try:
            await procesar_siguiente()
        except Exception:          # CancelledError NO entra aquí (3.8+)
            log.exception("fallo procesando")
        except BaseException:      # ...pero aquí sí, y rompe la cancelación
            log.exception("fallo procesando")
```

Si necesitas limpiar recursos al cancelar, hazlo y vuelve a lanzar la excepción. Nunca la absorbas:

```
async def consumidor(cola: asyncio.Queue):
    try:
        while True:
            mensaje = await cola.get()
            await manejar(mensaje)
    except asyncio.CancelledError:
        log.info("consumidor cancelado, cerrando limpiamente")
        raise                      # imprescindible: informa a quien cancela
    finally:
        await cerrar_recursos()
```

Hay una trampa adicional en el finally: si haces await de algo dentro de un bloque de limpieza mientras te están cancelando, esa espera también puede cancelarse. Para operaciones de cierre que deben completarse sí o sí, combina asyncio.shield() con su propio timeout:

```
async def transferir(origen, destino, importe):
    tx = await iniciar_transaccion(origen, destino, importe)
    try:
        await confirmar(tx)
    except asyncio.CancelledError:
        # La compensación debe ejecutarse aunque nos estén cancelando.
        async with asyncio.timeout(5):
            await asyncio.shield(revertir(tx))
        raise
```

shield se usa mal con facilidad. Protege a la tarea interna de la cancelación, pero el await externo sí se cancela: si no lo acompañas de un mecanismo que espere el resultado, acabas con trabajo ejecutándose sin nadie observándolo. Úsalo con moderación y solo para compensaciones o cierres.

Por último, el apagado. Un despliegue rutinario envía SIGTERM y espera; si no cancelas y esperas a tus tareas, te quedan mensajes a medio procesar y transacciones abiertas:

```
import asyncio, signal, contextlib

async def main():
    parar = asyncio.Event()
    loop = asyncio.get_running_loop()
    for sig in (signal.SIGTERM, signal.SIGINT):
        loop.add_signal_handler(sig, parar.set)

    async with asyncio.TaskGroup() as tg:
        tg.create_task(monitor_lag(), name="lag")
        servidor = tg.create_task(servir(), name="servidor")
        await parar.wait()
        servidor.cancel()          # TaskGroup espera a que termine de verdad

with contextlib.suppress(asyncio.CancelledError):
    asyncio.run(main())
```

6. Concurrencia acotada: lanzar 10.000 tareas no es escalar

Es tentador escribir tg.create_task() dentro de un bucle sobre una lista de 50.000 elementos. Asyncio te dejará hacerlo, y entonces abrirás 50.000 conexiones a la vez, tumbarás a tu proveedor, agotarás los descriptores de fichero del contenedor o te comerás la memoria del proceso.

La defensa mínima es un semáforo:

```
import asyncio

async def procesar_todo(http, urls: list[str], limite: int = 20) -> list[dict]:
    sem = asyncio.Semaphore(limite)

    async def uno(url: str) -> dict:
        async with sem:                      # como mucho "limite" a la vez
            r = await http.get(url)
            return r.json()

    async with asyncio.TaskGroup() as tg:
        tareas = [tg.create_task(uno(u), name=f"fetch:{u}") for u in urls]
    return [t.result() for t in tareas]
```

Funciona, pero fíjate en el matiz: el semáforo limita las llamadas en vuelo, no los objetos creados. Con 50.000 URLs sigues construyendo 50.000 objetos Task de golpe. Si la lista es grande o llega en streaming, el patrón correcto es una cola con un número fijo de trabajadores:

```
import asyncio

async def pipeline(fuente, n_workers: int = 20, cola_max: int = 100):
    cola: asyncio.Queue[str | None] = asyncio.Queue(maxsize=cola_max)

    async def worker(idx: int) -> None:
        while True:
            item = await cola.get()
            try:
                if item is None:             # señal de parada
                    return
                await procesar(item)
            except Exception:
                log.exception("item fallido", item=item)   # nunca escapa
            finally:
                cola.task_done()

    async with asyncio.TaskGroup() as tg:
        workers = [tg.create_task(worker(i), name=f"worker-{i}")
                   for i in range(n_workers)]
        async for item in fuente:            # maxsize aplica backpressure
            await cola.put(item)
        for _ in workers:
            await cola.put(None)
```

El maxsize de la cola es lo que aporta backpressure: cuando los trabajadores no dan abasto, cola.put() se bloquea y el productor deja de leer de la fuente. Sin ese límite, una cola sin tope es una fuga de memoria con otro nombre. Fíjate también en que el except está dentro del worker: si dejas que una excepción escape, pierdes un trabajador y el rendimiento cae de forma escalonada hasta que no queda ninguno.

7. Primitivas de sincronización: no mezcles mundos

Las primitivas de asyncio y las de threading se parecen y no son intercambiables. Un threading.Lock dentro de una corrutina bloquea el hilo del loop entero; un asyncio.Lock usado desde otro hilo no

protege absolutamente nada.

```
import asyncio

class TokenCache:
    def __init__(self) -> None:
        self._lock = asyncio.Lock()
        self._token: str | None = None
        self._expira: float = 0.0

    async def get(self) -> str:
        ahora = asyncio.get_running_loop().time()
        if self._token and ahora < self._expira:
            return self._token
        async with self._lock:                # evita renovaciones simultáneas
            ahora = asyncio.get_running_loop().time()
            if self._token and ahora < self._expira:
                return self._token            # otro ya lo renovó mientras esperábamos
            self._token, ttl = await pedir_token()
            self._expira = ahora + ttl * 0.9
            return self._token
```

La doble comprobación no es adorno: sin ella, todas las corrutinas que se quedaron esperando el lock renovarían el token una detrás de otra en cuanto lo obtengan. Es el mismo razonamiento del cache stampede, aplicado dentro de un proceso.

Si de verdad necesitas cruzar la frontera entre un hilo y el loop, la única forma segura es call_soon_threadsafe o run_coroutine_threadsafe:

```
import asyncio

def callback_de_libreria_sincrona(loop, dato):
    # Se ejecuta en un hilo ajeno: NO toques objetos de asyncio directamente.
    fut = asyncio.run_coroutine_threadsafe(publicar(dato), loop)
    fut.result(timeout=5)     # este result() es bloqueante, y es lo correcto aquí
```

8. Qué ha cambiado en Python 3.12, 3.13 y 3.14

Tres novedades recientes cambian decisiones prácticas de diseño:

- Tareas eager (3.12). Con loop.set_task_factory(asyncio.eager_task_factory), una corrutina empieza a ejecutarse de forma síncrona en el momento de construir la Task y solo se programa en el loop si llega a suspenderse. En cargas donde muchas corrutinas terminan sin bloquearse -- aciertos de caché, validaciones rápidas -- evita una vuelta completa al planificador y reduce la latencia de forma medible. Ojo con el cambio semántico: parte del trabajo ocurre antes de que create_task retorne.
- Introspección desde fuera (3.14). python -m asyncio ps <PID> lista las tareas vivas de un proceso ya en marcha, y python -m asyncio pstree <PID> las muestra como un árbol de llamadas con quién espera a quién. Para un servicio colgado en producción, esto sustituye horas de adivinar dónde se quedó parado, y no requiere haber instrumentado nada de antemano.
- Free-threading (3.14). asyncio ya tiene soporte de primera clase para el intérprete sin GIL, con la tarea actual almacenada en el estado del hilo y un loop por hilo. Abre la puerta a repartir varios event loops entre núcleos dentro de un mismo proceso, aunque a día de hoy sigue siendo una construcción opcional y con un ecosistema desigual: mide antes de apostar por ella.

```
# Tareas eager: actívalas si perfilas y ves ganancia, no por defecto.
import asyncio

async def main():
```

```
    asyncio.get_running_loop().set_task_factory(asyncio.eager_task_factory)
    await servir()

asyncio.run(main())
```

```
# En producción, para un proceso que se ha quedado colgado:
$ python -m asyncio pstree 4242

% % %    ( T )   T a s k - 1
        % % %    m a i n
                % % %    s e r v i r
                       % % %    ( T )   w o r k e r - 3
                              % % %    p r o c e s a r
                                     % % %    c o l a . g e t
```

Esa salida responde en un segundo la pregunta que más tiempo cuesta durante un incidente: ¿dónde están paradas mis tareas y quién las está esperando?

9. Errores frecuentes, en una lista

- Marcar como async def una función que por dentro es totalmente síncrona. No la hace concurrente; solo esconde el bloqueo detrás de una firma tranquilizadora.
- Crear un cliente HTTP o un pool de base de datos por petición en lugar de por proceso.
- Usar gather sin return_exceptions y dar por hecho que las tareas restantes se detienen.
- Capturar BaseException o CancelledError sin volver a lanzarla.
- Olvidar guardar la referencia de las tareas creadas con create_task.
- Colas sin maxsize: la fuga de memoria más educada que existe.
- Confiar en que asyncio.timeout te salvará de un bloqueo síncrono. No puede.
- Mezclar threading.Lock con corrutinas, o asyncio.Lock entre hilos.
- Apagar el proceso sin cancelar ni esperar a las tareas.
- Meterlo todo en el executor por defecto y convertir sus 32 hilos en el límite de concurrencia de tu servicio.

10. Checklist antes de desplegar

- Ningún import de librerías bloqueantes (requests, psycopg2 síncrono, time.sleep) dentro de rutas async.
- Métrica de lag del event loop publicada y con alerta configurada.
- Todas las llamadas de red con timeout explícito, y el presupuesto total propagado entre servicios.
- Concurrencia acotada con semáforo o pool de workers en cada fan-out.
- Tareas de fondo con referencia fuerte y logging de fallos.
- Manejo de SIGTERM que cancela tareas y espera un cierre limpio con plazo.
- Tests que ejerciten la cancelación, no solo el camino feliz.

Ese último punto es el que más se salta y el que más incidentes evita. Un test de cancelación es corto y atrapa bugs reales:

```
import asyncio, pytest

@pytest.mark.asyncio
async def test_consumidor_cierra_limpiamente():
    cola: asyncio.Queue = asyncio.Queue()
    tarea = asyncio.create_task(consumidor(cola))
    await asyncio.sleep(0)              # deja que la tarea arranque

    tarea.cancel()
    with pytest.raises(asyncio.CancelledError):
        await tarea

    assert recursos_cerrados()          # el bloque finally se ejecutó
```

Si ese test pasa, sabes dos cosas: que tu corrutina no se traga la cancelación y que su limpieza se ejecuta. Son justo las dos propiedades que necesitas durante un despliegue.

11. Ciclo de vida de los recursos: abre una vez, cierra siempre

Casi todos los servicios asyncio que se comportan mal en producción comparten un defecto estructural: crean sus clientes en el sitio equivocado. Un AsyncClient por petición tira el pool de conexiones en cada llamada; un pool de base de datos creado en el import se abre antes de que exista el event loop; un cliente que nadie cierra deja sockets en CLOSE_WAIT hasta que el kernel se queja. La solución no es más código, es un único lugar donde todo nace y muere.

Ese lugar es un gestor de contexto asíncrono. Y cuando tienes más de un recurso, la herramienta correcta es AsyncExitStack, que cierra en orden inverso y --esto es lo importante-- cierra también lo que ya se había abierto si el arranque falla a mitad.

```
# Ciclo de vida centralizado: un solo sitio donde se abre y se cierra todo.
import contextlib
from collections.abc import AsyncIterator

import asyncpg
import httpx
from fastapi import FastAPI


@contextlib.asynccontextmanager
async def lifespan(app: FastAPI) -> AsyncIterator[None]:
    async with contextlib.AsyncExitStack() as stack:
        app.state.http = await stack.enter_async_context(
            httpx.AsyncClient(
                timeout=httpx.Timeout(5.0, connect=2.0),
                limits=httpx.Limits(max_connections=100, max_keepalive_connections=20),
            )
        )
        app.state.db = await stack.enter_async_context(
            asyncpg.create_pool(dsn=DSN, min_size=5, max_size=20, command_timeout=5.0)
        )
        yield
    # Al salir, AsyncExitStack cierra en orden inverso. Si create_pool hubiera
    # fallado, el cliente HTTP ya abierto se cierra igualmente: sin fugas.


app = FastAPI(lifespan=lifespan)
```

Fíjate en el detalle que casi nadie escribe: los recursos se crean dentro de una corrutina, no a nivel de módulo. Un pool de asyncpg construido en el import se ata al loop que exista en ese momento --o a ninguno-- y luego falla con un desconcertante attached to a different loop en cuanto el servidor arranca el suyo. La regla práctica: en el ámbito del módulo solo van constantes y configuración; todo lo que tenga un socket detrás se construye en el lifespan.

El mismo patrón sirve fuera de FastAPI. Un worker de cola, un consumidor de Kafka o un script largo pueden envolver su cuerpo entero en un AsyncExitStack y ganar exactamente la misma garantía. Si tu servicio tiene tareas de fondo, mete también el TaskGroup en la pila: al cerrarse la aplicación, el grupo cancela y espera a sus tareas antes de que se cierren los clientes que esas tareas usan. El orden importa, y AsyncExitStack lo respeta gratis.

12. Pools de conexiones: cuando quedarse sin conexiones parece un cuelgue

Hay un incidente que se repite en todos los servicios asyncio que hablan con una base de datos, y es especialmente cruel porque no produce errores. Las peticiones empiezan a tardar. Luego tardan más. El CPU está bajo, la base de datos está tranquila, los logs no dicen nada. Lo que está pasando es que el pool está agotado y cada corrutina nueva se queda esperando su turno en una cola invisible.

La causa es aritmética. Si tu pool tiene 20 conexiones y una consulta lenta ocupa cada una durante 400 ms, tu techo real son 50 consultas por segundo, independientemente de cuántas tareas lances. El semáforo de la sección 6 limita las llamadas concurrentes; el pool limita algo distinto y más bajo. Cuando los dos números no cuadran, gana el más pequeño y lo hace en silencio.

La primera defensa es negarse a esperar indefinidamente:

```
import asyncpg


async def obtener_usuario(pool: asyncpg.Pool, user_id: int):
    # Sin timeout, acquire() espera para siempre cuando el pool está lleno:
    # el servicio "se cuelga" sin registrar un solo error.
    async with pool.acquire(timeout=2.0) as conn:
        return await conn.fetchrow(
            "SELECT id, email FROM usuarios WHERE id = $1", user_id
        )
```

Un TimeoutError al adquirir es una señal utilísima: te dice que el cuello de botella está en el pool y no en la base de datos. Sin él, el mismo problema se te presenta como latencia difusa. Y con él puedes devolver un 503 rápido en lugar de acumular peticiones que ya nadie va a leer.

La segunda defensa es medir. Un pool sin métricas es una caja negra:

```
import asyncpg


def metricas_del_pool(pool: asyncpg.Pool) -> dict[str, int]:
    tamano = pool.get_size()
    libres = pool.get_idle_size()
    return {
        "abiertas": tamano,
        "libres": libres,
        "en_uso": tamano - libres,
        "maximo": pool.get_max_size(),
    }
```

Exporta en_uso / maximo como gauge y alerta cuando pase de 0,8 de forma sostenida. Esa alerta llega minutos antes que la de latencia y apunta directamente a la causa.

Sobre el dimensionado, la intuición engaña: un pool más grande no es mejor. Cada conexión de PostgreSQL es un proceso del servidor con su propia memoria, y pasado cierto punto añadir conexiones reduce el rendimiento total por contención de locks y de planificador. Multiplica siempre

max_size × número de procesos × número de réplicas y compáralo con max_connections; si te acercas, la respuesta no es subir max_connections sino poner un pooler como PgBouncer delante. Un pool pequeño con una cola corta y visible casi siempre rinde mejor que uno grande que esconde el problema.

Lo mismo aplica a HTTP. httpx.Limits(max_connections=100) es un pool: si tu semáforo permite 200 llamadas concurrentes al mismo host, 100 de ellas estarán esperando conexión y su timeout de lectura correrá mientras tanto. Alinea los dos números o acepta que los timeouts que verás no miden lo que crees que miden.

13. Generadores asíncronos: streaming sin fugas

Los generadores asíncronos son la forma más elegante de procesar algo que no cabe en memoria: lees una fila, la procesas, la sueltas. Traen backpressure de regalo, porque el productor no avanza hasta que el consumidor pide el siguiente elemento. Y tienen una trampa que casi nadie ve venir.

La trampa es el break. Cuando abandonas un generador a medias, su bloque finally no se ejecuta en ese momento: queda pendiente de que alguien llame a aclose() o de que el recolector de basura lo alcance. Si dentro del finally hay una conexión de base de datos, esa conexión se queda ocupada un tiempo indeterminado. Con tráfico suficiente, has convertido un break inocente en el agotamiento de pool de la sección anterior.

```
import contextlib
from collections.abc import AsyncIterator

import asyncpg


async def leer_pedidos(pool: asyncpg.Pool) -> AsyncIterator[asyncpg.Record]:
    async with pool.acquire() as conn, conn.transaction():
        # Cursor del servidor: no trae la tabla entera a memoria.
        async for fila in conn.cursor("SELECT id, total FROM pedidos ORDER BY id"):
            yield fila


async def total_hasta(pool: asyncpg.Pool, tope: float) -> float:
    total = 0.0
    # aclosing garantiza que el generador se cierra -y suelta la conexión-
    # aunque salgamos con break o con una excepción.
    async with contextlib.aclosing(leer_pedidos(pool)) as pedidos:
        async for fila in pedidos:
            total += float(fila["total"])
            if total >= tope:
                break
    return total
```

contextlib.aclosing existe desde Python 3.10 y es la respuesta corta a todo esto. Úsalo siempre que un async for pueda terminar antes de tiempo. Si controlas el arranque del programa, asyncio.run() ya llama a loop.shutdown_asyncgens() al salir, lo que cierra los generadores que queden vivos; pero eso ocurre al final del proceso, demasiado tarde para liberar una conexión que necesitabas hace diez minutos, y no ocurre en absoluto si montas el loop a mano.

Hay un segundo uso de los generadores asíncronos que merece la pena: agrupar en lotes. Muchos sistemas de destino son mucho más rápidos escribiendo de cien en cien que de uno en uno, y un generador intermedio expresa esa transformación sin ensuciar la lógica de negocio.

```
import asyncio
from collections.abc import AsyncIterator


async def en_lotes(fuente: AsyncIterator[dict], tam: int = 100, espera: float = 0.5):
    """Agrupa elementos en lotes de `tam`, o cierra el lote tras `espera`
    segundos sin llenarlo. Evita que el último puñado se quede colgado."""
    lote: list[dict] = []
    it = fuente.__aiter__()
    while True:
        try:
            elemento = await asyncio.wait_for(it.__anext__(), timeout=espera)
        except asyncio.TimeoutError:
            if lote:
                yield lote
                lote = []
            continue
        except StopAsyncIteration:
            break
        lote.append(elemento)
        if len(lote) >= tam:
            yield lote
            lote = []
    if lote:
        yield lote
```

El temporizador es la parte que se olvida. Sin él, un lote a medio llenar espera indefinidamente al elemento noventa y nueve, y en un sistema con tráfico irregular eso significa datos que llegan con horas de retraso durante la noche. Con él, el peor caso de latencia está acotado por espera.

14. Carreras y hedging: patrones más allá de TaskGroup

TaskGroup resuelve el caso dominante --lanzar N cosas y esperarlas todas--, pero hay un caso que no cubre: quiero la primera que responda y no me importan las demás. Ahí entra asyncio.wait con FIRST_COMPLETED, y con él una responsabilidad que la API no te recuerda: las tareas que no ganan siguen vivas. Si no las cancelas, siguen consumiendo conexiones y, dentro de un TaskGroup, impiden que el bloque termine.

El patrón útil de verdad es el hedging: si la primera petición tarda más de lo normal, lanza una segunda en paralelo y quédate con la que llegue antes. Es la técnica estándar para recortar la cola de latencia (p99) a cambio de un poco de tráfico extra.

```
import asyncio
from collections.abc import Awaitable, Callable


async def con_hedge(
    peticion: Callable[[], Awaitable[bytes]],
    retraso: float = 0.05,
) -> bytes:
    """Lanza una segunda petición si la primera no responde en `retraso`.
    Solo para lecturas idempotentes: duplicar una escritura duplica el efecto."""
    async with asyncio.TaskGroup() as tg:
        primera = tg.create_task(peticion(), name="hedge-1")

        hechas, _ = await asyncio.wait({primera}, timeout=retraso)
        if hechas:
            return primera.result()

        segunda = tg.create_task(peticion(), name="hedge-2")
        hechas, pendientes = await asyncio.wait(
            {primera, segunda}, return_when=asyncio.FIRST_COMPLETED
        )
        ganadora = hechas.pop()
        for perdedora in pendientes:
```

```
        perdedora.cancel()   # sin esto, el TaskGroup esperaría a la lenta
    return ganadora.result()
```

Tres avisos antes de que lo copies. Uno: solo lecturas idempotentes. Hacer hedging de un cobro es duplicar cobros. Dos: el retraso debe salir de tus datos, no de tu intuición; el valor razonable ronda el p95 de esa llamada, de modo que solo se dispare en el 5 % peor. Tres: si la ganadora terminó con excepción, ganadora.result() la relanza; considera esperar también a la otra antes de rendirte.

El primo pobre de este patrón es asyncio.as_completed, útil cuando quieres ir procesando resultados según llegan sin esperar al último. Desde Python 3.13 acepta también un argumento timeout y devuelve las tareas originales, lo que hace mucho más cómodo saber cuál terminó. Sigue siendo tu responsabilidad cancelar lo que quede si decides salir antes.

15. Reintentos que caben en el presupuesto

Los reintentos y los timeouts se diseñan juntos o no se diseñan. Un reintento no es gratis: multiplica el tiempo total y multiplica la carga sobre un servicio que probablemente ya está sufriendo. Tres intentos con un timeout de dos segundos cada uno son seis segundos de espera para el cliente, más los backoffs. Si tu SLA de respuesta es de tres segundos, ese código no cumple aunque cada intento individual parezca razonable.

La forma correcta es restar del presupuesto propagado que ya montaste en la sección 4:

```
import asyncio
import random

REINTENTABLES = (asyncio.TimeoutError, ConnectionError, OSError)


async def con_reintentos(fn, intentos: int = 3, base: float = 0.1):
    for intento in range(intentos):
        restante = tiempo_restante()          # presupuesto de la sección 4
        if restante <= 0:
            raise TimeoutError("presupuesto agotado antes de reintentar")

        try:
            async with asyncio.timeout(min(restante, 2.0)):
                return await fn()
        except asyncio.CancelledError:
            raise                             # cancelación no es un fallo
        except REINTENTABLES:
            if intento == intentos - 1:
                raise
            espera = random.uniform(0, base * 2 ** intento)   # full jitter
            if espera >= tiempo_restante():
                raise TimeoutError("no queda presupuesto para el backoff")
            await asyncio.sleep(espera)
```

Los detalles que hacen que esto funcione: el except asyncio.CancelledError antes del genérico, porque una cancelación externa no es un error que debas reintentar; el full jitter, que evita que mil clientes reintenten en el mismo milisegundo; y la comprobación de que el propio backoff cabe en lo que queda, que es lo que impide dormir dos segundos para luego descubrir que ya no había tiempo.

Un apunte de escala: si todos tus clientes reintentan tres veces, una caída parcial se convierte en un pico de tráfico del 300 % justo cuando el servicio menos lo aguanta. Por eso los sistemas grandes usan presupuestos de reintento --como máximo un 10 % de reintentos sobre el total de peticiones-- y un circuit breaker que corta del todo cuando la tasa de fallo se dispara. Reintentar más no es reintentar mejor.

16. Tareas de fondo supervisadas

Todo servicio real acaba teniendo trabajo perpetuo: un consumidor de cola, un refresco de caché, un heartbeat. La versión ingenua es asyncio.create_task(bucle()) en el arranque y a otra cosa. Funciona hasta la primera excepción no prevista, momento en el que la tarea muere en silencio --porque nadie hace await de ella-- y el servicio sigue en pie, aparentemente sano, sin consumir nada de la cola. Es uno de los fallos más caros de diagnosticar porque el síntoma aparece aguas abajo, horas después.

La respuesta es un supervisor: un bucle que reinicia el trabajo cuando cae, con backoff para no montar un bucle de fuego rápido si el fallo es permanente.

```
import asyncio
import logging

log = logging.getLogger(__name__)


async def supervisar(nombre: str, fabrica, base: float = 1.0, tope: float = 30.0):
    """Reinicia `fabrica()` cuando cae. Nunca vuelve por su cuenta:
    solo termina cuando se le cancela desde fuera."""
    espera = base
    while True:
        try:
            await fabrica()
            espera = base                # terminó limpio: reinicia el backoff
        except asyncio.CancelledError:
            log.info("supervisor %s cancelado", nombre)
            raise                        # nunca te tragues la cancelación
        except Exception:
            log.exception("%s cayó; reintento en %.1fs", nombre, espera)
            await asyncio.sleep(espera)
            espera = min(espera * 2, tope)
```

El raise tras capturar CancelledError es obligatorio. Sin él, el supervisor se traga el apagado y tu terminationGracePeriodSeconds se convierte en un SIGKILL. Y fíjate en que el backoff se reinicia cuando la tarea termina limpiamente: un consumidor que procesa un lote y vuelve no debe acumular penalización.

Dónde colgar el supervisor: dentro del TaskGroup del lifespan, no suelto. Así el apagado de la aplicación lo cancela y espera a que termine antes de cerrar el pool que el supervisor está usando. Un supervisor huérfano que sigue vivo mientras se cierra la base de datos produce una traza espectacular y completamente inútil en los logs de despliegue.

Un matiz de diseño crash-only: no intentes que la tarea de fondo se recupere de todo internamente. Es mucho más simple --y más fiable-- dejarla morir y reiniciarla desde arriba, siempre que el trabajo sea idempotente o retome desde un offset persistido. La recuperación fina dentro de un bucle de mil líneas es donde viven los bugs que solo aparecen a las tres de la mañana.

17. El puente síncrono: llamar a corrutinas desde código que no lo es

Antes o después aparece la biblioteca heredada que no sabe nada de asyncio, o el comando de gestión de Django que necesita llamar a tu cliente asíncrono. El primer instinto es asyncio.run(mi_corrutina()), y si ya hay un loop corriendo en ese hilo obtienes RuntimeError: asyncio.run() cannot be called from a running event loop.

La respuesta que circula por internet es nest_asyncio. Evítala. Parchea el loop para permitir reentrada, algo que asyncio no garantiza, y lo que consigues es que una corrutina pueda ejecutarse en medio de otra con el estado a medio actualizar. Los bugs que produce son no deterministas y aparecen en producción, no en tu portátil.

Hay tres respuestas legítimas, según de dónde vengas. Si eres código síncrono en el hilo principal y no hay loop, asyncio.run() es correcto: úsalo una sola vez, en el borde del programa. Si eres código síncrono dentro de una corrutina, no cruces la frontera: extrae la parte bloqueante y mándala con asyncio.to_thread(). Y si eres código síncrono en otro hilo que necesita hablar con un loop que ya existe, el puente correcto es este:

```
import asyncio
import threading


class PuenteAsincrono:
    """Un event loop propio en un hilo de fondo, para código síncrono
    heredado que necesita llamar a corrutinas sin montar y tirar un loop
    en cada invocación."""

    def __init__(self) -> None:
        self._loop = asyncio.new_event_loop()
        self._hilo = threading.Thread(
            target=self._loop.run_forever, name="puente-asyncio", daemon=True
        )
        self._hilo.start()

    def ejecutar(self, coro, timeout: float | None = None):
        # run_coroutine_threadsafe es la única API de asyncio pensada para
        # llamarse desde otro hilo. Devuelve un concurrent.futures.Future.
        fut = asyncio.run_coroutine_threadsafe(coro, self._loop)
        return fut.result(timeout)      # bloquea al llamante, no al loop

    def cerrar(self) -> None:
        self._loop.call_soon_threadsafe(self._loop.stop)
        self._hilo.join(timeout=5)
        self._loop.close()
```

Un loop de larga vida en un hilo de fondo tiene una ventaja concreta sobre asyncio.run() repetido: los pools de conexiones sobreviven entre llamadas. Con asyncio.run() por invocación, cada llamada abre y cierra su cliente HTTP, y el coste del handshake TLS te come el beneficio entero.

18. Observabilidad: el lag del loop como métrica de primera clase

En un servicio síncrono, la métrica que te avisa de que algo va mal es el uso de CPU o el tamaño del pool de hilos. En asyncio no existe ese equivalente: un proceso al 100 % de un núcleo puede estar perfectamente sano, y uno al 15 % puede estar completamente atascado. La métrica que de verdad te dice cómo está tu servicio es el lag del event loop: cuánto se retrasa el loop respecto a lo que había prometido hacer.

Medirlo es trivial: duermes un intervalo conocido y compruebas cuánto has dormido de verdad. La diferencia es tiempo que el loop pasó sin poder atenderte, es decir, tiempo que alguien pasó bloqueándolo.

```
import asyncio
import time

from prometheus_client import Gauge, Histogram

LAG = Histogram(
```

```
    "event_loop_lag_seconds",
    "Retraso entre el despertar previsto y el real",
    buckets=(0.001, 0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 5.0),
)
TAREAS = Gauge("asyncio_tareas_vivas", "Tareas asyncio sin terminar")


async def monitor_del_loop(intervalo: float = 0.25) -> None:
    while True:
        inicio = time.perf_counter()
        await asyncio.sleep(intervalo)
        LAG.observe(time.perf_counter() - inicio - intervalo)
        TAREAS.set(len(asyncio.all_tasks()))
```

Un histograma, no una media: el lag es una distribución con cola larga y la media la esconde. En un servicio sano el p99 vive por debajo de 10 ms. Si el p99 sube de 100 ms, tienes código bloqueante en el camino caliente, y la alerta debería dispararse ahí, no cuando la latencia de usuario ya se ha degradado. Lanza el monitor desde el lifespan, dentro del TaskGroup, y ponle nombre.

El gauge de tareas vivas es el complemento: una línea que sube y no baja es una fuga de tareas, casi siempre un create_task cuyo resultado nadie recoge. Vale la pena vigilar también el crecimiento del contador respecto al tráfico; si las peticiones son planas y las tareas crecen, tienes trabajo acumulándose.

Dos hábitos más que cuestan poco y valen mucho. El primero, nombrar las tareas: tg.create_task(procesar(p), name=f"procesar-pedido-{{}p.id{}}"). Ese nombre es lo que verás en python -m asyncio ps y en los avisos de tareas destruidas mientras estaban pendientes; sin él, la salida es una lista de objetos anónimos y el diagnóstico se vuelve arqueología. El segundo, entender cómo viajan las contextvars: se copian al crear la tarea, no al esperarla. Por eso un request_id o un span de OpenTelemetry sobreviven a create_task, y por eso no llegan solos a un run_in_executor, donde tienes que pasarlos explícitamente con contextvars.copy_context().run(...).

19. Testear código asíncrono sin tests que tardan y mienten

Los tests de código asíncrono fallan de dos maneras características: se cuelgan para siempre en CI, o pasan siempre porque están llenos de sleep que enmascaran las condiciones de carrera que deberían detectar. Ambas se arreglan con configuración y disciplina.

Empieza por el modo. pytest-asyncio usa strict por defecto, que exige marcar cada test con @pytest.mark.asyncio y decorar las fixtures con @pytest_asyncio.fixture. Es verboso, pero es el modo correcto si convives con anyio o trio, porque los plugins no se pisan. Si tu proyecto es asyncio puro, auto elimina ese ruido. Elige uno explícitamente: dejarlo implícito es la causa habitual del misterioso async def functions are not natively supported, donde el test no se ejecuta y pytest lo marca como pasado.

```
# pyproject.toml
# [tool.pytest.ini_options]
# asyncio_mode = "auto"                        # o "strict" si usas anyio/trio
# asyncio_default_fixture_loop_scope = "function"
# timeout = 30                                 # pytest-timeout: CI nunca se cuelga

import asyncio

import pytest


async def test_la_limpieza_ocurre_al_cancelar():
```

```
    limpiado = asyncio.Event()

    async def trabajo():
        try:
            await asyncio.sleep(3600)
        except asyncio.CancelledError:
            limpiado.set()
            raise                   # imprescindible: propaga la cancelación

    tarea = asyncio.create_task(trabajo())
    await asyncio.sleep(0)          # cede el control: deja que llegue al await
    tarea.cancel()

    with pytest.raises(asyncio.CancelledError):
        await tarea
    assert limpiado.is_set()
```

Ese test es más valioso de lo que parece: comprueba a la vez que tu corrutina limpia al cancelar y que vuelve a lanzar la cancelación, que es el error de la sección 5. Escribe uno así por cada recurso que tengas que soltar.

Para sincronizar, usa primitivas en lugar de relojes. await asyncio.sleep(0.1) confiando en que el otro lado ya habrá llegado es la receta de un test que falla una vez de cada cincuenta en CI, en la máquina más lenta, un viernes. asyncio.Event, asyncio.Barrier (3.11+) o simplemente await asyncio.sleep(0) para ceder un ciclo del loop expresan la intención sin depender del tiempo real. Y cuando de verdad necesites que pase el tiempo, un reloj falso como el de anyio.pytest_plugin o time-machine hace que el test dure milisegundos.

Por último, mide lo que importa: un test que verifica que tu handler no bloquea el loop. Lanza el handler y, en paralelo, un asyncio.sleep corto repetido; si el lag observado supera un umbral, el test falla. Es el único tipo de test que detecta la regresión de alguien que añadió una llamada síncrona a una función async def.

20. Despliegue: procesos, loops y hacia dónde va la API

Un proceso asyncio usa un núcleo. Esa frase resume casi toda la estrategia de despliegue. Para aprovechar una máquina de ocho núcleos necesitas ocho procesos, cada uno con su propio event loop, detrás de un balanceador o de un socket compartido --que es exactamente lo que hacen uvicorn --workers, gunicorn con workers uvicorn, o N réplicas en Kubernetes. Añadir hilos dentro del proceso no multiplica el rendimiento del loop; solo te da un sitio donde aparcar trabajo bloqueante.

Sobre el loop en sí, uvloop sigue siendo la mejora más barata que existe: es una implementación en Cython sobre libuv, entra como sustituto directo y en cargas dominadas por E/S de red suele dar mejoras claras sin tocar tu código. Desde la 0.22 hay ruedas para CPython 3.14, incluida la variante free-threading. Mídelo en tu carga antes de darlo por bueno: si tu cuello de botella es la base de datos, el loop no era el problema.

La forma de instalarlo ha cambiado y conviene actualizarse. El sistema de políticas de event loop (asyncio.set_event_loop_policy) quedó deprecado en Python 3.15 y está previsto que desaparezca en 3.16. El reemplazo es asyncio.Runner con loop_factory, disponible desde 3.11:

```
import asyncio

try:
    import uvloop
    fabrica_de_loop = uvloop.new_event_loop
```

```
except ImportError:              # sin uvloop, el loop estándar sirve
    fabrica_de_loop = asyncio.new_event_loop


def main() -> None:
    # asyncio.Runner (3.11+) sustituye a las políticas de event loop,
    # deprecadas en 3.15 y previstas para eliminación en 3.16.
    with asyncio.Runner(loop_factory=fabrica_de_loop) as runner:
        runner.run(servidor())


if __name__ == "__main__":
    main()
```

Merece la pena mencionar también asyncio.iscoroutinefunction, deprecado en 3.15 a favor de inspect.iscoroutinefunction. Si tienes código de infraestructura que decide qué hacer según si algo es corrutina --decoradores, despachadores de tareas--, ese cambio te toca.

Y la novedad estructural: la 3.14 trae soporte de primera clase para free-threading en asyncio, con una lista enlazada por hilo para las tareas nativas que además ha traído mejoras del 10-20 % en los benchmarks estándar y menos memoria. Con un build free-threaded puedes correr varios event loops en hilos distintos del mismo proceso y escalar de forma aproximadamente lineal. No es todavía el despliegue por defecto --el ecosistema de extensiones en C sigue poniéndose al día--, pero cambia el horizonte: la pregunta "¿un proceso por núcleo o hilos?" deja de tener una única respuesta obvia.

21. Caso práctico: un servicio de ingesta completo

Juntemos todo. El servicio lee mensajes de una cola, para cada uno llama a una API externa y escribe el resultado en PostgreSQL. Tiene concurrencia acotada, presupuesto de tiempo, apagado limpio y métricas. Son unas sesenta líneas, y cada decisión sale de una sección anterior.

```
import asyncio
import contextlib
import logging
import signal

import asyncpg
import httpx

log = logging.getLogger(__name__)
CONCURRENCIA = 20


async def procesar(msg: dict, http: httpx.AsyncClient, pool: asyncpg.Pool) -> None:
    async with asyncio.timeout(5.0):                    # presupuesto por mensaje
        r = await http.get(f"/enriquecer/{'{'}msg['id']{'}'}")
        r.raise_for_status()
        datos = r.json()

    async with pool.acquire(timeout=2.0) as conn:       # nunca esperar sin límite
        await conn.execute(
            "INSERT INTO eventos (id, payload) VALUES ($1, $2)"
            " ON CONFLICT (id) DO NOTHING",             # idempotente: se puede reintentar
            msg["id"], datos,
        )


async def trabajador(n: int, cola: asyncio.Queue, http, pool) -> None:
    while True:
        msg = await cola.get()
        try:
            await procesar(msg, http, pool)
        except asyncio.CancelledError:
            await cola.put(msg)                         # devuélvelo, no lo pierdas
```

```
            raise
        except Exception:
            log.exception("mensaje %s falló", msg.get("id"))
        finally:
            cola.task_done()


async def main() -> None:
    parar = asyncio.Event()
    loop = asyncio.get_running_loop()
    for sig in (signal.SIGTERM, signal.SIGINT):
        loop.add_signal_handler(sig, parar.set)

    async with contextlib.AsyncExitStack() as stack:
        http = await stack.enter_async_context(
            httpx.AsyncClient(base_url=API, timeout=httpx.Timeout(4.0, connect=2.0))
        )
        pool = await stack.enter_async_context(
            asyncpg.create_pool(dsn=DSN, min_size=5, max_size=CONCURRENCIA)
        )
        cola: asyncio.Queue = asyncio.Queue(maxsize=CONCURRENCIA * 2)  # backpressure

        async with asyncio.TaskGroup() as tg:
            for n in range(CONCURRENCIA):
                tg.create_task(trabajador(n, cola, http, pool), name=f"worker-{n}")
            productor = tg.create_task(alimentar(cola, parar), name="productor")

            await parar.wait()
            log.info("SIGTERM recibido; drenando")
            productor.cancel()
            await cola.join()               # termina lo que ya está en vuelo
            raise asyncio.CancelledError    # cancela a los trabajadores


if __name__ == "__main__":
    with contextlib.suppress(asyncio.CancelledError):
        asyncio.run(main())
```

Las decisiones que importan, en orden. El maxsize de la cola es lo que impide que un productor rápido llene la memoria; es el mismo backpressure de la sección 6. El ON CONFLICT DO NOTHING hace la escritura idempotente, que es lo que permite reintentar y devolver mensajes a la cola sin duplicar. El await cola.put(msg) dentro del manejador de CancelledError --seguido de raise-- es la diferencia entre un apagado que pierde trabajo y uno que no. Y el await cola.join() antes de cancelar es lo que convierte el apagado en un drenado: los mensajes que ya estaban en vuelo terminan, los nuevos no entran.

El tamaño del pool y la concurrencia son el mismo número a propósito. Si pusieras CONCURRENCIA = 100 con un pool de 20, tendrías ochenta trabajadores esperando conexión con su timeout de escritura corriendo: exactamente el fallo de la sección 12, ahora con la ventaja de que sabrías reconocerlo.

22. Runbook: el servicio no responde

Son las tres de la mañana y el servicio está vivo pero no contesta. Esta es la secuencia que lleva al diagnóstico en minutos en lugar de horas.

- Mira el lag del loop antes que nada. Si el p99 está por las nubes, alguien está bloqueando el hilo y el problema es CPU o una llamada síncrona. Si el lag está sano y la latencia es alta, el loop está libre y estás esperando a algo externo: pool, red o un lock.
- Pide el árbol de tareas. python -m asyncio pstree PID (3.14+) te dibuja el grafo de llamadas de todas las tareas, en todos los hilos, sin instrumentar nada y sin parar el proceso. Si cincuenta tareas están esperando en la misma línea, ya tienes la respuesta. python -m asyncio ps PID da la misma información en tabla, más cómoda para filtrar.

• Si es un build sin esa herramienta, py-spy dump --pid PID te da las pilas de los hilos. Un hilo principal parado dentro de una función que no es select ni epoll_wait es tu bloqueo.

• Revisa las métricas del pool. en_uso == maximo sostenido confirma agotamiento. Comprueba a la vez las consultas lentas del lado de la base de datos: casi siempre hay una consulta que se ha degradado y está ocupando conexiones el triple de tiempo.

• Busca el lock. Si el árbol muestra tareas esperando en Lock.acquire, alguien mantiene el lock mientras hace E/S. Ese es el patrón: un lock tomado alrededor de una llamada de red serializa todo el servicio.

• Mitiga y luego arregla. Reiniciar devuelve el servicio, pero si no capturaste el árbol de tareas antes, has borrado la prueba. Un volcado de pstree cuesta un segundo y es la diferencia entre arreglarlo hoy o esperar a la siguiente vez.

Un consejo que ahorra mucho: guarda ese volcado en el ticket. Los cuelgues de asyncio se parecen mucho entre sí en los gráficos y son muy distintos en el árbol de tareas, que es donde está la información.

23. Preguntas frecuentes

¿asyncio es más rápido que los hilos? Para miles de conexiones lentas, sí, y por un margen amplio: una tarea cuesta kilobytes, un hilo cuesta megabytes de pila. Para cien conexiones y consultas rápidas, la diferencia es pequeña y los hilos son más fáciles de escribir. Y para trabajo intensivo de CPU, asyncio no aporta nada: sigue siendo un solo hilo.

¿Debo migrar todo mi servicio síncrono a asyncio? Rara vez. La migración parcial es peor que no migrar, porque una sola llamada síncrona en el camino caliente anula el beneficio de todo lo demás. Migra servicios enteros orientados a E/S, o no migres.

¿Qué uso, asyncio.timeout() o asyncio.wait_for()? asyncio.timeout() desde 3.11. Es un gestor de contexto, se compone con otros timeouts anidados y expresa un plazo sobre un bloque en lugar de sobre una sola llamada. wait_for sigue funcionando y ahora está implementado por encima de aquél.

¿Puedo usar time.sleep() en una corrutina si es muy poco? No. "Muy poco" multiplicado por tu tasa de peticiones deja de ser poco muy deprisa, y el coste no lo paga esa petición: lo pagan todas las demás que estaban esperando su turno en el loop.

¿Por qué mi excepción no aparece en los logs? Casi seguro es una tarea de fuego y olvido cuyo resultado nadie recoge. Usa TaskGroup, o guarda la referencia y añade un add_done_callback que registre el fallo. Es la sección 3, y es el bug más común de todos.

¿Cuántas tareas concurrentes puedo lanzar? La pregunta correcta es cuántas puede absorber tu dependencia más lenta. El límite no lo pone asyncio --aguanta decenas de miles-- sino el pool de base de datos, el rate limit de la API o la memoria de los payloads en vuelo. Empieza por ese número y ponle un semáforo.

¿Merece la pena uvloop? Si tu servicio es sobre todo E/S de red y estás cerca del techo, sí, y es un cambio de dos líneas. Si tu p99 está dominado por una consulta de 300 ms, el loop no era tu problema.

Para cerrar

asyncio no es difícil por su sintaxis, que se aprende en una tarde. Es difícil porque su modelo de ejecución hace que ciertos errores no fallen, sino que degraden. Un servicio que bloquea el loop no lanza excepciones: solo va más lento. Una tarea recolectada por el GC no deja traza: simplemente no ocurre. Una cancelación tragada no rompe nada hasta el día que un despliegue tarda diez minutos en terminar.

Instrumenta el lag del loop, acota la concurrencia en cada fan-out, pon timeouts en todas partes con un presupuesto que se propague, y trata la cancelación como parte del contrato de cada corrutina que escribas. Con esas cuatro cosas cubres la inmensa mayoría de los incidentes que verás en un servicio asyncio.

PuigFlows

Technical guide · Real-World Projects

asyncio in Production: Never Block the Event Loop -- TaskGroups, Cancellation and Bounded Concurrency

A practical asyncio guide for Python services in production: why blocking the event loop degrades the whole process without raising a single exception, how to catch it by measuring loop lag and with Python 3.14 introspection, structured concurrency with TaskGroup and handling ExceptionGroup via except*, the task the garbage collector makes vanish, timeouts with a deadline budget propagated across services, correct cancellation with cleanup and shield, bounded concurrency with semaphores and backpressured queues, synchronization primitives, and what changes with eager tasks, python -m asyncio pstree and free-threading. With production-ready code and a deployment checklist.

Level: Intermediate · 8,234 words · 37 min read

2026-08-16 · puigflows.com

A practical asyncio guide for services that are already in production. The code targets Python 3.12 or later; wherever something needs a specific version, I say so explicitly.

The mental model: one thread, many tasks, zero interruptions

An asyncio program runs on a single thread. The event loop keeps a queue of ready tasks, runs one until that task yields control with await, and then moves on to the next. There is no preemption: nobody takes the CPU away from anybody. That property is exactly what lets you serve thousands of slow connections with a ridiculously small memory footprint, and it is also what makes a single slip freeze the entire service.

From that follows the one rule that actually matters: never block the event loop thread. Everything else in this guide is a corollary.

1. The original sin: blocking the event loop

The most expensive asyncio bug is not the one that blows up with an exception. It is the one that degrades latency across the whole process without leaving a trace in your logs. Look at this endpoint, which looks asynchronous:

```
# ANTI-PATTERN: requests is synchronous. It blocks the whole thread.
import requests
from fastapi import FastAPI

app = FastAPI()
```

```
@app.get("/profile/{user_id}")
async def profile(user_id: str):
    r = requests.get(f"https://internal.api/users/{user_id}", timeout=5)
    return r.json()
```

The signature says async def, but requests.get() sits on the socket without yielding. For those 300 ms -- or those 5 seconds when the dependency misbehaves -- the event loop cannot serve any other request. At 50 requests per second, the p99 latency of completely unrelated endpoints spikes, and the chart gives you no clue why.

The correct version uses an async client and, crucially, reuses it:

```
# CORRECT: async client with a process-wide connection pool.
import httpx
from contextlib import asynccontextmanager
from fastapi import FastAPI

@asynccontextmanager
async def lifespan(app: FastAPI):
    # One client for the whole process: reuses TCP and TLS connections.
    app.state.http = httpx.AsyncClient(
        timeout=httpx.Timeout(5.0, connect=2.0),
        limits=httpx.Limits(max_connections=100, max_keepalive_connections=20),
    )
    yield
    await app.state.http.aclose()

app = FastAPI(lifespan=lifespan)

@app.get("/profile/{user_id}")
async def profile(user_id: str):
    r = await app.state.http.get(f"https://internal.api/users/{user_id}")
    r.raise_for_status()
    return r.json()
```

Creating a fresh AsyncClient per request is a common, silent mistake: you lose keep-alive, you pay a TLS handshake on every call, and under load you eventually exhaust ephemeral ports.

The blocking that does not look like blocking

Synchronous network calls are easy to spot in code review. These are not:

- Synchronous database drivers used inside a coroutine: psycopg2, the Django ORM in sync mode, SQLAlchemy without its asyncio engine.
- Reading and writing files with open(). Disk feels instantaneous until the network volume degrades.
- Pure CPU work: parsing a 40 MB JSON payload, compressing, hashing passwords with bcrypt or Argon2, rendering a PDF.
- time.sleep() instead of asyncio.sleep(), usually hidden inside an inherited retry helper.
- Synchronous DNS resolution in libraries that do not delegate it to the loop.
- A logging handler that writes over the network without buffering.

How to catch it before your users do

The loop debug mode warns when a callback takes too long. It is the cheapest way to find blocking in development and staging:

```
import asyncio, logging

async def main():
    loop = asyncio.get_running_loop()
    loop.slow_callback_duration = 0.1   # warn if anything holds the loop >100 ms
    await serve()

logging.basicConfig(level=logging.DEBUG)
asyncio.run(main(), debug=True)

# Typical output:
# Executing <Task pending name='Task-7' coro=<profile()>> took 1.402 seconds
```

For production, a much better guardrail is to measure the loop lag directly: schedule a wake-up every 250 ms and check how far it drifts from the plan. On a healthy loop the drift is microseconds; when something is blocking it, the number explodes.

```
import asyncio, time
from prometheus_client import Gauge

LAG = Gauge("event_loop_lag_seconds", "Event loop scheduling delay")

async def monitor_lag(interval: float = 0.25) -> None:
    while True:
        start = time.perf_counter()
        await asyncio.sleep(interval)
        LAG.set((time.perf_counter() - start) - interval)
```

Alert when that metric stays above roughly 50 ms. It is by far the signal that points fastest at the real culprit when the complaint is "everything is slow and we do not know why".

When blocking is unavoidable: threads

Sometimes there is no async alternative: a vendor SDK, an old library, a CPU-bound function. That is what asyncio.to_thread() is for -- it runs the function on the default executor and hands control back to the loop meanwhile.

```
import asyncio
from argon2 import PasswordHasher

ph = PasswordHasher()

async def verify_password(stored_hash: str, password: str) -> bool:
    # Argon2 burns CPU on purpose: keep it off the event loop.
    return await asyncio.to_thread(ph.verify, stored_hash, password)
```

Two details that usually cost an incident:

- The default executor is small. The default ThreadPoolExecutor uses min(32, os.cpu_count() + 4) threads. If you funnel every blocking call through it, that number becomes your real concurrency ceiling and work queues up silently. Create dedicated executors per kind of work so one bottleneck does not contaminate the rest.
- Threads do not fix CPU. With the GIL enabled, CPU-heavy work still serializes. That is what ProcessPoolExecutor is for -- or better, move the computation out of the web process entirely.

```
import asyncio
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor

io_pool = ThreadPoolExecutor(max_workers=16, thread_name_prefix="io")
cpu_pool = ProcessPoolExecutor(max_workers=4)
```

```
async def read_legacy(path: str) -> bytes:
    loop = asyncio.get_running_loop()
    return await loop.run_in_executor(io_pool, _read_sync, path)

async def render_pdf(data: dict) -> bytes:
    loop = asyncio.get_running_loop()
    # ProcessPool: arguments and return value must be picklable.
    return await loop.run_in_executor(cpu_pool, _render_heavy, data)
```

And one poorly documented detail that bites: the future returned by run_in_executor cannot really be cancelled once the function has started. If you cancel the await, your coroutine moves on but the thread keeps working to completion. Design those functions to be short, or to check a stop flag.

2. TaskGroup: the modern replacement for gather

For years, launching concurrent work in asyncio meant asyncio.gather(). It works, but its error behaviour is treacherous: with the default settings, if one coroutine fails, gather propagates that exception immediately and leaves the rest running. Those orphaned tasks keep holding connections and may end up writing to your database long after you returned a 500 to the client.

asyncio.TaskGroup (Python 3.11+) implements structured concurrency: the async with block does not exit until every child task has finished, and if one fails, the others are cancelled automatically.

```
import asyncio

async def load_dashboard(http, user_id: str) -> dict:
    async with asyncio.TaskGroup() as tg:
        t_profile = tg.create_task(get_profile(http, user_id), name="profile")
        t_orders  = tg.create_task(get_orders(http, user_id),  name="orders")
        t_alerts  = tg.create_task(get_alerts(http, user_id),  name="alerts")
    # On exiting the block, all three have completed successfully. Guaranteed.
    return {
        "profile": t_profile.result(),
        "orders":  t_orders.result(),
        "alerts":  t_alerts.result(),
    }
```

Notice where the results are read: after the block. Inside the async with, the tasks may still be unfinished. The name= parameter on a TaskGroup create_task is available from Python 3.14 and is worth using, because it shows up in tracebacks and in the introspection tooling.

When something fails, TaskGroup raises an ExceptionGroup, because several tasks may have failed at once. You handle it with except*:

```
try:
    data = await load_dashboard(http, user_id)
except* httpx.TimeoutException as eg:
    # eg.exceptions holds every timeout that occurred
    log.warning("dashboard timeouts", n=len(eg.exceptions))
    raise HTTPException(504, "Upstream too slow")
except* PermissionError:
    raise HTTPException(403, "Forbidden")
```

Here is the trap: a plain except Exception does not catch an ExceptionGroup by its contents, only by the group type itself. This is the classic mistake when migrating from gather to TaskGroup -- your existing error handling quietly stops working and exceptions bubble unfiltered to the edge of the application.

Is there still a place for gather? Yes, in one specific case: when you want partial results even if

something fails.

```
# Aggregate view where partial failure is acceptable.
results = await asyncio.gather(
    get_profile(http, uid),
    get_orders(http, uid),
    get_recommendations(http, uid),
    return_exceptions=True,
)
profile, orders, recos = [
    None if isinstance(r, BaseException) else r for r in results
]
```

With return_exceptions=True the exceptions come back as values, so you have to inspect them. If you do not, a failure becomes an exception object travelling through your business logic until it triggers a baffling AttributeError three layers down.

3. The task that vanishes

This bug quietly kills background work and is almost impossible to reproduce locally, because it depends on the garbage collector.

```
# ANTI-PATTERN: this task can disappear mid-execution.
async def handler(event):
    asyncio.create_task(send_metrics(event))   # nobody keeps the reference
    return {"ok": True}
```

The event loop only keeps a weak reference to tasks. If nothing else holds a strong reference, the collector may destroy the task before it finishes. In development it works every time; under load, an unpredictable share of your metrics, emails or webhooks simply never goes out. And there is no error to search for in the logs.

The fix is to hold the reference and clean it up on completion:

```
_background_tasks: set[asyncio.Task] = set()

def spawn_background(coro, *, name: str) -> asyncio.Task:
    task = asyncio.create_task(coro, name=name)
    _background_tasks.add(task)                    # strong reference
    task.add_done_callback(_background_tasks.discard)
    task.add_done_callback(_log_if_failed)
    return task

def _log_if_failed(task: asyncio.Task) -> None:
    if task.cancelled():
        return
    exc = task.exception()
    if exc is not None:
        log.error("background task failed",
                  task=task.get_name(), exc_info=exc)
```

That second callback solves another real problem: a task that fails without anyone calling result() only produces an "exception was never retrieved" warning when the object is destroyed -- sometimes minutes later, with no useful context. Logging the error explicitly turns a mystery into an actionable log line.

One design note: for background work that must survive a restart, the right answer is not an asyncio task but a persistent queue. An in-memory task dies with the process, and deploys are frequent.

4. Timeouts: the one you forgot is the one that takes you down

A coroutine without a timeout can wait forever. In a service with bounded concurrency, a handful of hung tasks drains the pool and the outage looks like saturation rather than a missing timeout. The root cause hides behind the symptom.

Since Python 3.11, asyncio.timeout() is the preferred tool because it works as a context manager and covers whole blocks, not a single await:

```
import asyncio

async def get_price(http, sku: str) -> float:
    async with asyncio.timeout(2.0):
        r = await http.get(f"/prices/{sku}")
        data = r.json()
        return await enrich(data)         # the budget covers the WHOLE block
```

When the deadline expires, asyncio cancels the task from the inside and turns that cancellation into a TimeoutError. The difference from asyncio.wait_for() is ergonomics: wait_for wraps a single await, while timeout() wraps a sequence of operations.

In distributed systems, what you actually want is not a per-call timeout but a deadline that propagates. If you promise the user a response in 3 seconds, every downstream hop should respect what is left of that budget instead of starting its own:

```
import asyncio, contextvars, time

_deadline: contextvars.ContextVar[float | None] = contextvars.ContextVar(
    "deadline", default=None
)

def remaining(default: float) -> float:
    d = _deadline.get()
    return default if d is None else max(0.0, d - time.monotonic())

async def with_budget(seconds: float, coro):
    _deadline.set(time.monotonic() + seconds)
    async with asyncio.timeout(seconds):
        return await coro

async def call_inventory(http, sku: str):
    # Never spends more than the global budget has left.
    async with asyncio.timeout(min(1.5, remaining(1.5))):
        return await http.get(f"/inventory/{sku}")
```

Always use time.monotonic() for deadlines. The wall clock can jump backwards on an NTP adjustment and leave you with negative -- or worse, eternal -- timeouts.

And remember the fundamental limitation: asyncio.timeout() does not cover blocking code. If a synchronous call eats the thread, the loop cannot even run the timer. The timeout fires once the blocking ends, which is exactly when it is no longer useful.

5. Cancellation: the part almost everyone gets wrong

Cancellation in asyncio is implemented by raising asyncio.CancelledError inside the coroutine at its next await point. Since Python 3.8 that exception inherits from BaseException rather than Exception, precisely so a careless except Exception does not swallow it.

```
# ANTI-PATTERN: turns a cancellable task into one that never dies.
async def worker():
    while True:
        try:
            await process_next()
        except Exception:         # CancelledError does NOT land here (3.8+)
            log.exception("processing failed")
        except BaseException:     # ...but it lands here, and breaks cancellation
            log.exception("processing failed")
```

If you need to clean up on cancellation, do it and re-raise. Never absorb it:

```
async def consumer(queue: asyncio.Queue):
    try:
        while True:
            message = await queue.get()
            await handle(message)
    except asyncio.CancelledError:
        log.info("consumer cancelled, shutting down cleanly")
        raise                       # essential: tell the canceller it worked
    finally:
        await close_resources()
```

There is an extra trap in that finally: if you await something inside a cleanup block while you are being cancelled, that await can be cancelled too. For shutdown operations that must complete, combine asyncio.shield() with its own timeout:

```
async def transfer(source, target, amount):
    tx = await begin_transaction(source, target, amount)
    try:
        await commit(tx)
    except asyncio.CancelledError:
        # The compensation must run even while we are being cancelled.
        async with asyncio.timeout(5):
            await asyncio.shield(rollback(tx))
        raise
```

shield is easy to misuse. It protects the inner task from cancellation, but the outer await is still cancelled: unless you pair it with something that waits for the result, you end up with work running that nobody is observing. Use it sparingly, and only for compensation or shutdown paths.

Finally, shutdown itself. A routine deploy sends SIGTERM and waits; if you do not cancel and await your tasks, you are left with half-processed messages and open transactions:

```
import asyncio, signal, contextlib

async def main():
    stop = asyncio.Event()
    loop = asyncio.get_running_loop()
    for sig in (signal.SIGTERM, signal.SIGINT):
        loop.add_signal_handler(sig, stop.set)

    async with asyncio.TaskGroup() as tg:
        tg.create_task(monitor_lag(), name="lag")
        server = tg.create_task(serve(), name="server")
        await stop.wait()
        server.cancel()            # TaskGroup waits for it to really finish

with contextlib.suppress(asyncio.CancelledError):
    asyncio.run(main())
```

6. Bounded concurrency: firing 10,000 tasks is not scaling

It is tempting to write tg.create_task() inside a loop over a list of 50,000 items. Asyncio will let you, and then you will open 50,000 connections at once, take down your provider, exhaust the container file descriptors, or eat the process memory.

The minimum defence is a semaphore:

```
import asyncio

async def process_all(http, urls: list[str], limit: int = 20) -> list[dict]:
    sem = asyncio.Semaphore(limit)

    async def one(url: str) -> dict:
        async with sem:                      # at most "limit" in flight
            r = await http.get(url)
            return r.json()

    async with asyncio.TaskGroup() as tg:
        tasks = [tg.create_task(one(u), name=f"fetch:{u}") for u in urls]
    return [t.result() for t in tasks]
```

It works, but note the nuance: the semaphore bounds the calls in flight, not the objects created. With 50,000 URLs you still build 50,000 Task objects up front. If the list is large or arrives as a stream, the right pattern is a queue with a fixed number of workers:

```
import asyncio

async def pipeline(source, n_workers: int = 20, queue_max: int = 100):
    queue: asyncio.Queue[str | None] = asyncio.Queue(maxsize=queue_max)

    async def worker(idx: int) -> None:
        while True:
            item = await queue.get()
            try:
                if item is None:             # stop signal
                    return
                await process(item)
            except Exception:
                log.exception("item failed", item=item)   # never escapes
            finally:
                queue.task_done()

    async with asyncio.TaskGroup() as tg:
        workers = [tg.create_task(worker(i), name=f"worker-{i}")
                   for i in range(n_workers)]
        async for item in source:            # maxsize applies backpressure
            await queue.put(item)
        for _ in workers:
            await queue.put(None)
```

The queue maxsize is what gives you backpressure: when the workers cannot keep up, queue.put() blocks and the producer stops reading from the source. Without that bound, an unbounded queue is a memory leak with a friendlier name. Note too that the except sits inside the worker: if you let an exception escape, you lose a worker and throughput drops in steps until none are left.

7. Synchronization primitives: do not mix worlds

The asyncio primitives and the threading ones look alike and are not interchangeable. A threading.Lock inside a coroutine blocks the entire loop thread; an asyncio.Lock used from another thread protects absolutely nothing.

```
import asyncio

class TokenCache:
    def __init__(self) -> None:
        self._lock = asyncio.Lock()
        self._token: str | None = None
        self._expires: float = 0.0

    async def get(self) -> str:
        now = asyncio.get_running_loop().time()
        if self._token and now < self._expires:
            return self._token
        async with self._lock:                # prevents concurrent refreshes
            now = asyncio.get_running_loop().time()
            if self._token and now < self._expires:
                return self._token            # someone refreshed while we waited
            self._token, ttl = await fetch_token()
            self._expires = now + ttl * 0.9
            return self._token
```

The double check is not decoration: without it, every coroutine that queued on the lock would refresh the token one after another as soon as it acquires it. Same reasoning as a cache stampede, applied inside a single process.

If you genuinely need to cross the boundary between a thread and the loop, the only safe way is call_soon_threadsafe or run_coroutine_threadsafe:

```
import asyncio

def callback_from_sync_library(loop, payload):
    # Runs on a foreign thread: do NOT touch asyncio objects directly.
    fut = asyncio.run_coroutine_threadsafe(publish(payload), loop)
    fut.result(timeout=5)     # this result() blocks, and that is correct here
```

8. What changed in Python 3.12, 3.13 and 3.14

Three recent additions change practical design decisions:

- Eager tasks (3.12). With loop.set_task_factory(asyncio.eager_task_factory), a coroutine starts executing synchronously at Task construction time and is only scheduled on the loop if it actually suspends. In workloads where many coroutines finish without blocking -- cache hits, quick validations -- this avoids a full round trip through the scheduler and measurably cuts latency. Mind the semantic change: some of the work happens before create_task returns.
- Introspection from outside (3.14). python -m asyncio ps <PID> lists the live tasks of an already running process, and python -m asyncio pstree <PID> renders them as a call tree showing who is awaiting whom. For a hung service in production, this replaces hours of guessing where it got stuck, and it requires no upfront instrumentation.
- Free-threading (3.14). asyncio now has first-class support for the GIL-free interpreter, with the current task stored in the thread state and one loop per thread. It opens the door to spreading several event loops across cores inside a single process -- though today it is still an optional build with an uneven ecosystem, so measure before betting on it.

```
# Eager tasks: enable them if profiling shows a win, not by default.
import asyncio

async def main():
    asyncio.get_running_loop().set_task_factory(asyncio.eager_task_factory)
    await serve()
```

```
asyncio.run(main())
```

```
# In production, for a process that has hung:
$ python -m asyncio pstree 4242

% % %    ( T )   T a s k - 1
        % % %    m a i n
                % % %    s e r v e
                        % % %    ( T )   w o r k e r - 3
                                % % %    p r o c e s s
                                        % % %    q u e u e . g e t
```

That output answers, in one second, the question that costs the most time during an incident: where are my tasks parked, and who is waiting on them?

9. Common mistakes, as a list

- Marking a function async def when its body is entirely synchronous. It does not make it concurrent; it just hides the blocking behind a reassuring signature.
- Creating an HTTP client or a database pool per request instead of per process.
- Using gather without return_exceptions and assuming the remaining tasks stop.
- Catching BaseException or CancelledError without re-raising.
- Forgetting to keep a reference to tasks created with create_task.
- Queues without maxsize: the most polite memory leak there is.
- Trusting asyncio.timeout to save you from synchronous blocking. It cannot.
- Mixing threading.Lock with coroutines, or asyncio.Lock across threads.
- Shutting the process down without cancelling and awaiting your tasks.
- Funnelling everything into the default executor, turning its 32 threads into your service concurrency ceiling.

10. Pre-deployment checklist

- No imports of blocking libraries (requests, sync psycopg2, time.sleep) inside async paths.
- Event loop lag metric exported, with an alert configured.
- Every network call has an explicit timeout, and the overall budget propagates across services.
- Bounded concurrency via semaphore or worker pool at every fan-out.
- Background tasks with a strong reference and failure logging.
- SIGTERM handling that cancels tasks and waits for a clean, time-boxed shutdown.
- Tests that exercise cancellation, not only the happy path.

That last point is the most skipped and the one that prevents the most incidents. A cancellation test is short and catches real bugs:

```
import asyncio, pytest

@pytest.mark.asyncio
async def test_consumer_shuts_down_cleanly():
```

```
    queue: asyncio.Queue = asyncio.Queue()
    task = asyncio.create_task(consumer(queue))
    await asyncio.sleep(0)              # let the task start

    task.cancel()
    with pytest.raises(asyncio.CancelledError):
        await task

    assert resources_closed()           # the finally block ran
```

If that test passes, you know two things: your coroutine does not swallow cancellation, and its cleanup runs. Those are exactly the two properties you need during a deploy.

11. Resource lifecycle: open once, always close

Almost every asyncio service that misbehaves in production shares one structural defect: it creates its clients in the wrong place. An AsyncClient per request throws away the connection pool on every call; a database pool created at import time is built before an event loop exists; a client nobody closes leaves sockets in CLOSE_WAIT until the kernel complains. The fix is not more code, it is a single place where everything is born and dies.

That place is an async context manager. And when you have more than one resource, the right tool is AsyncExitStack, which closes in reverse order and -- this is the part that matters -- also closes whatever was already opened if startup fails halfway through.

```
# Centralized lifecycle: one place where everything opens and closes.
import contextlib
from collections.abc import AsyncIterator

import asyncpg
import httpx
from fastapi import FastAPI


@contextlib.asynccontextmanager
async def lifespan(app: FastAPI) -> AsyncIterator[None]:
    async with contextlib.AsyncExitStack() as stack:
        app.state.http = await stack.enter_async_context(
            httpx.AsyncClient(
                timeout=httpx.Timeout(5.0, connect=2.0),
                limits=httpx.Limits(max_connections=100, max_keepalive_connections=20),
            )
        )
        app.state.db = await stack.enter_async_context(
            asyncpg.create_pool(dsn=DSN, min_size=5, max_size=20, command_timeout=5.0)
        )
        yield
    # On exit, AsyncExitStack closes in reverse order. Had create_pool failed,
    # the already-open HTTP client would still be closed: no leaks.


app = FastAPI(lifespan=lifespan)
```

Note the detail hardly anyone writes down: resources are created inside a coroutine, not at module level. An asyncpg pool built at import time binds to whatever loop exists at that moment -- or to none -- and then fails with a baffling attached to a different loop as soon as the server starts its own. The practical rule: module scope holds constants and configuration only; anything with a socket behind it is built in the lifespan.

The same pattern works outside FastAPI. A queue worker, a Kafka consumer or a long-running script can wrap their entire body in an AsyncExitStack and get exactly the same guarantee. If your service has

background tasks, put the TaskGroup on the stack too: when the app shuts down, the group cancels and awaits its tasks before the clients those tasks use get closed. Order matters, and AsyncExitStack respects it for free.

12. Connection pools: when running out of connections looks like a hang

There is one incident that recurs in every asyncio service talking to a database, and it is especially cruel because it produces no errors. Requests start taking longer. Then longer. CPU is low, the database is calm, the logs say nothing. What is happening is that the pool is exhausted and every new coroutine is waiting its turn in an invisible queue.

The cause is arithmetic. If your pool has 20 connections and a slow query holds each one for 400 ms, your real ceiling is 50 queries per second, no matter how many tasks you launch. The semaphore from section 6 caps concurrent calls; the pool caps something different and lower. When the two numbers disagree, the smaller one wins, silently.

The first defense is refusing to wait forever:

```
import asyncpg


async def get_user(pool: asyncpg.Pool, user_id: int):
    # Without a timeout, acquire() waits forever when the pool is full:
    # the service "hangs" without logging a single error.
    async with pool.acquire(timeout=2.0) as conn:
        return await conn.fetchrow(
            "SELECT id, email FROM users WHERE id = $1", user_id
        )
```

A TimeoutError on acquire is an extremely useful signal: it tells you the bottleneck is the pool and not the database. Without it, the same problem shows up as diffuse latency. With it, you can return a fast 503 instead of piling up requests nobody is going to read anymore.

The second defense is measuring. A pool without metrics is a black box:

```
import asyncpg


def pool_metrics(pool: asyncpg.Pool) -> dict[str, int]:
    size = pool.get_size()
    idle = pool.get_idle_size()
    return {
        "open": size,
        "idle": idle,
        "in_use": size - idle,
        "max": pool.get_max_size(),
    }
```

Export in_use / max as a gauge and alert when it stays above 0.8. That alert fires minutes before the latency one and points straight at the cause.

On sizing, intuition misleads: a bigger pool is not a better pool. Every PostgreSQL connection is a server-side process with its own memory, and past a certain point adding connections lowers total throughput through lock and planner contention. Always multiply max_size × processes × replicas and compare against max_connections; if you are close, the answer is not raising max_connections but putting a pooler like PgBouncer in front. A small pool with a short, visible queue almost always beats a

large one that hides the problem.

The same applies to HTTP. httpx.Limits(max_connections=100) is a pool: if your semaphore allows 200 concurrent calls to the same host, 100 of them will be waiting for a connection with their read timeout ticking. Align the two numbers, or accept that the timeouts you see are not measuring what you think they measure.

13. Async generators: streaming without leaks

Async generators are the most elegant way to process something that does not fit in memory: read a row, process it, drop it. They give you backpressure for free, because the producer does not advance until the consumer asks for the next item. And they have a trap almost nobody sees coming.

The trap is break. When you abandon a generator midway, its finally block does not run at that moment: it waits for someone to call aclose() or for the garbage collector to reach it. If that finally holds a database connection, that connection stays checked out for an indeterminate amount of time. With enough traffic, you have turned an innocent break into the pool exhaustion of the previous section.

```
import contextlib
from collections.abc import AsyncIterator

import asyncpg


async def read_orders(pool: asyncpg.Pool) -> AsyncIterator[asyncpg.Record]:
    async with pool.acquire() as conn, conn.transaction():
        # Server-side cursor: does not pull the whole table into memory.
        async for row in conn.cursor("SELECT id, total FROM orders ORDER BY id"):
            yield row


async def total_up_to(pool: asyncpg.Pool, cap: float) -> float:
    total = 0.0
    # aclosing guarantees the generator is closed -and releases the connection-
    # even if we leave via break or an exception.
    async with contextlib.aclosing(read_orders(pool)) as orders:
        async for row in orders:
            total += float(row["total"])
            if total >= cap:
                break
    return total
```

contextlib.aclosing has been available since Python 3.10 and is the short answer to all of this. Use it whenever an async for can end early. If you control program startup, asyncio.run() already calls loop.shutdown_asyncgens() on the way out, which closes generators still alive; but that happens at the end of the process, far too late to free a connection you needed ten minutes ago, and it does not happen at all if you build the loop by hand.

There is a second use of async generators worth knowing: batching. Many destination systems are far faster writing a hundred rows at a time than one at a time, and an intermediate generator expresses that transformation without polluting your business logic.

```
import asyncio
from collections.abc import AsyncIterator


async def in_batches(source: AsyncIterator[dict], size: int = 100, wait: float = 0.5):
    """Group items into batches of `size`, or flush after `wait` seconds
    without filling one. Keeps the last few items from getting stranded."""
    batch: list[dict] = []
```

```
    it = source.__aiter__()
    while True:
        try:
            item = await asyncio.wait_for(it.__anext__(), timeout=wait)
        except asyncio.TimeoutError:
            if batch:
                yield batch
                batch = []
            continue
        except StopAsyncIteration:
            break
        batch.append(item)
        if len(batch) >= size:
            yield batch
            batch = []
    if batch:
        yield batch
```

The timer is the part people forget. Without it, a half-full batch waits indefinitely for item ninety-nine, and in a system with uneven traffic that means data arriving hours late overnight. With it, worst-case latency is bounded by wait.

14. Racing and hedging: patterns beyond TaskGroup

TaskGroup solves the dominant case -- launch N things and await them all -- but there is one case it does not cover: I want the first one that answers and I do not care about the rest. That is where asyncio.wait with FIRST_COMPLETED comes in, and with it a responsibility the API does not remind you about: the tasks that do not win are still running. If you do not cancel them, they keep holding connections and, inside a TaskGroup, they keep the block from finishing.

The genuinely useful pattern here is hedging: if the first request takes longer than usual, fire a second one in parallel and keep whichever answers first. It is the standard technique for trimming tail latency (p99) in exchange for a little extra traffic.

```
import asyncio
from collections.abc import Awaitable, Callable


async def with_hedge(
    request: Callable[[], Awaitable[bytes]],
    delay: float = 0.05,
) -> bytes:
    """Fire a second request if the first does not answer within `delay`.
    Idempotent reads only: duplicating a write duplicates the effect."""
    async with asyncio.TaskGroup() as tg:
        first = tg.create_task(request(), name="hedge-1")

        done, _ = await asyncio.wait({first}, timeout=delay)
        if done:
            return first.result()

        second = tg.create_task(request(), name="hedge-2")
        done, pending = await asyncio.wait(
            {first, second}, return_when=asyncio.FIRST_COMPLETED
        )
        winner = done.pop()
        for loser in pending:
            loser.cancel()      # without this, the TaskGroup waits for the slow one
        return winner.result()
```

Three warnings before you copy it. One: idempotent reads only. Hedging a payment means charging twice. Two: the delay should come from your data, not your intuition; a sensible value sits around the p95 of that call, so it only fires for the worst 5 %. Three: if the winner finished with an exception,

winner.result() re-raises it; consider awaiting the other one before giving up.

The poor cousin of this pattern is asyncio.as_completed, useful when you want to process results as they arrive rather than waiting for the last one. Since Python 3.13 it also accepts a timeout argument and yields the original tasks back, which makes it far easier to know which one finished. Cancelling whatever remains if you bail out early is still your job.

15. Retries that fit inside the budget

Retries and timeouts are designed together or they are not designed at all. A retry is not free: it multiplies total time and multiplies load on a service that is probably already struggling. Three attempts with a two-second timeout each is six seconds of waiting for the client, plus backoffs. If your response SLA is three seconds, that code does not meet it even though each individual attempt looks reasonable.

The right shape is to subtract from the propagated budget you already built in section 4:

```
import asyncio
import random

RETRYABLE = (asyncio.TimeoutError, ConnectionError, OSError)


async def with_retries(fn, attempts: int = 3, base: float = 0.1):
    for attempt in range(attempts):
        remaining = time_remaining()          # budget from section 4
        if remaining <= 0:
            raise TimeoutError("budget exhausted before retrying")

        try:
            async with asyncio.timeout(min(remaining, 2.0)):
                return await fn()
        except asyncio.CancelledError:
            raise                             # cancellation is not a failure
        except RETRYABLE:
            if attempt == attempts - 1:
                raise
            sleep_for = random.uniform(0, base * 2 ** attempt)   # full jitter
            if sleep_for >= time_remaining():
                raise TimeoutError("no budget left for backoff")
            await asyncio.sleep(sleep_for)
```

The details that make this work: the except asyncio.CancelledError before the generic one, because an external cancellation is not an error you should retry; full jitter, which stops a thousand clients from retrying in the same millisecond; and the check that the backoff itself fits in what remains, which is what stops you from sleeping two seconds only to discover there was no time left.

A note on scale: if all your clients retry three times, a partial outage becomes a 300 % traffic spike exactly when the service can least handle it. That is why large systems use retry budgets -- at most 10 % retries over total requests -- plus a circuit breaker that cuts off entirely when the failure rate spikes. Retrying more is not retrying better.

16. Supervised background tasks

Every real service ends up with perpetual work: a queue consumer, a cache refresh, a heartbeat. The naive version is asyncio.create_task(loop()) at startup and move on. It works until the first unexpected exception, at which point the task dies silently -- because nobody awaits it -- and the service stays up, apparently healthy, consuming nothing from the queue. It is one of the most expensive failures to

diagnose because the symptom appears downstream, hours later.

The answer is a supervisor: a loop that restarts the work when it dies, with backoff so a permanent failure does not turn into a hot restart loop.

```
import asyncio
import logging

log = logging.getLogger(__name__)


async def supervise(name: str, factory, base: float = 1.0, cap: float = 30.0):
    """Restart `factory()` when it dies. Never returns on its own:
    it only ends when cancelled from outside."""
    delay = base
    while True:
        try:
            await factory()
            delay = base                  # finished cleanly: reset the backoff
        except asyncio.CancelledError:
            log.info("supervisor %s cancelled", name)
            raise                         # never swallow cancellation
        except Exception:
            log.exception("%s crashed; retrying in %.1fs", name, delay)
            await asyncio.sleep(delay)
            delay = min(delay * 2, cap)
```

The raise after catching CancelledError is mandatory. Without it, the supervisor swallows your shutdown and terminationGracePeriodSeconds turns into a SIGKILL. And note that the backoff resets when the task finishes cleanly: a consumer that processes a batch and returns should not accumulate a penalty.

Where to hang the supervisor: inside the lifespan TaskGroup, not loose. That way application shutdown cancels it and waits for it to finish before closing the pool the supervisor is using. An orphaned supervisor still running while the database closes produces a spectacular and completely useless traceback in your deploy logs.

One crash-only design note: do not try to make the background task recover from everything internally. It is far simpler -- and more reliable -- to let it die and restart it from above, as long as the work is idempotent or resumes from a persisted offset. Fine-grained recovery inside a thousand-line loop is where the bugs that only show up at 3 a.m. live.

17. The sync bridge: calling coroutines from code that is not async

Sooner or later the legacy library that knows nothing about asyncio shows up, or the Django management command that needs to call your async client. The first instinct is asyncio.run(my_coroutine()), and if a loop is already running in that thread you get RuntimeError: asyncio.run() cannot be called from a running event loop.

The answer circulating online is nest_asyncio. Avoid it. It patches the loop to allow reentrancy, something asyncio does not guarantee, and what you get is one coroutine running in the middle of another with half-updated state. The bugs it produces are non-deterministic and show up in production, not on your laptop.

There are three legitimate answers depending on where you are coming from. If you are sync code on the main thread with no loop, asyncio.run() is correct: use it once, at the edge of the program. If you are sync code inside a coroutine, do not cross the boundary: extract the blocking part and send it off with asyncio.to_thread(). And if you are sync code on another thread that needs to talk to a loop that already

exists, this is the correct bridge:

```
import asyncio
import threading


class AsyncBridge:
    """A dedicated event loop on a background thread, for legacy sync code
    that needs to call coroutines without spinning up and tearing down a
    loop on every invocation."""

    def __init__(self) -> None:
        self._loop = asyncio.new_event_loop()
        self._thread = threading.Thread(
            target=self._loop.run_forever, name="asyncio-bridge", daemon=True
        )
        self._thread.start()

    def run(self, coro, timeout: float | None = None):
        # run_coroutine_threadsafe is the only asyncio API designed to be
        # called from another thread. It returns a concurrent.futures.Future.
        fut = asyncio.run_coroutine_threadsafe(coro, self._loop)
        return fut.result(timeout)      # blocks the caller, not the loop

    def close(self) -> None:
        self._loop.call_soon_threadsafe(self._loop.stop)
        self._thread.join(timeout=5)
        self._loop.close()
```

A long-lived loop on a background thread has a concrete advantage over repeated asyncio.run(): connection pools survive between calls. With asyncio.run() per invocation, every call opens and closes its HTTP client, and the TLS handshake cost eats the entire benefit.

18. Observability: loop lag as a first-class metric

In a synchronous service, the metric that warns you something is wrong is CPU usage or thread pool size. In asyncio there is no equivalent: a process pinning 100 % of a core can be perfectly healthy, and one at 15 % can be completely stuck. The metric that actually tells you how your service is doing is event loop lag: how far behind the loop is relative to what it promised to do.

Measuring it is trivial: sleep for a known interval and check how long you actually slept. The difference is time the loop spent unable to serve you -- that is, time someone spent blocking it.

```
import asyncio
import time

from prometheus_client import Gauge, Histogram

LAG = Histogram(
    "event_loop_lag_seconds",
    "Delay between scheduled and actual wakeup",
    buckets=(0.001, 0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 5.0),
)
TASKS = Gauge("asyncio_live_tasks", "Unfinished asyncio tasks")


async def loop_monitor(interval: float = 0.25) -> None:
    while True:
        start = time.perf_counter()
        await asyncio.sleep(interval)
        LAG.observe(time.perf_counter() - start - interval)
        TASKS.set(len(asyncio.all_tasks()))
```

A histogram, not an average: lag is a long-tailed distribution and the mean hides it. In a healthy service

the p99 lives below 10 ms. If the p99 goes above 100 ms, you have blocking code on the hot path, and the alert should fire there, not once user-facing latency has already degraded. Launch the monitor from the lifespan, inside the TaskGroup, and give it a name.

The live-tasks gauge is the complement: a line that goes up and never comes down is a task leak, almost always a create_task whose result nobody collects. It is also worth watching its growth relative to traffic; if requests are flat and tasks are climbing, work is piling up.

Two more habits that cost little and pay a lot. First, name your tasks: tg.create_task(process(o), name=f"process-order-{{}o.id{}}"). That name is what you will see in python -m asyncio ps and in the "Task was destroyed but it is pending" warnings; without it the output is a list of anonymous objects and diagnosis turns into archaeology. Second, understand how contextvars travel: they are copied when the task is created, not when it is awaited. That is why a request_id or an OpenTelemetry span survives create_task, and why it does not reach a run_in_executor on its own -- there you have to pass it explicitly with contextvars.copy_context().run(...).

19. Testing async code without slow tests that lie

Async tests fail in two characteristic ways: they hang forever in CI, or they always pass because they are full of sleep calls masking the race conditions they were supposed to catch. Both are fixed with configuration and discipline.

Start with the mode. pytest-asyncio defaults to strict, which requires marking every test with @pytest.mark.asyncio and decorating fixtures with @pytest_asyncio.fixture. It is verbose, but it is the correct mode if you coexist with anyio or trio, because the plugins do not step on each other. If your project is pure asyncio, auto removes that noise. Pick one explicitly: leaving it implicit is the usual cause of the mysterious async def functions are not natively supported, where the test never runs and pytest marks it as passed.

```
# pyproject.toml
# [tool.pytest.ini_options]
# asyncio_mode = "auto"                        # or "strict" if you use anyio/trio
# asyncio_default_fixture_loop_scope = "function"
# timeout = 30                                 # pytest-timeout: CI never hangs

import asyncio

import pytest


async def test_cleanup_runs_on_cancel():
    cleaned = asyncio.Event()

    async def work():
        try:
            await asyncio.sleep(3600)
        except asyncio.CancelledError:
            cleaned.set()
            raise                       # essential: propagate the cancellation

    task = asyncio.create_task(work())
    await asyncio.sleep(0)              # yield control: let it reach the await
    task.cancel()

    with pytest.raises(asyncio.CancelledError):
        await task
    assert cleaned.is_set()
```

That test is more valuable than it looks: it checks both that your coroutine cleans up on cancellation and

that it re-raises the cancellation, which is the mistake from section 5. Write one like it for every resource you have to release.

For synchronization, use primitives instead of clocks. await asyncio.sleep(0.1) trusting that the other side has arrived by now is the recipe for a test that fails one time in fifty in CI, on the slowest machine, on a Friday. asyncio.Event, asyncio.Barrier (3.11+) or simply await asyncio.sleep(0) to yield one loop cycle express the intent without depending on wall time. And when you genuinely need time to pass, a fake clock like anyio's pytest plugin or time-machine makes the test run in milliseconds.

Finally, test what matters: a test that verifies your handler does not block the loop. Run the handler and, in parallel, a short repeated asyncio.sleep; if observed lag exceeds a threshold, the test fails. It is the only kind of test that catches the regression from whoever added a synchronous call inside an async def.

20. Deployment: processes, loops, and where the API is headed

One asyncio process uses one core. That sentence summarizes most of your deployment strategy. To use an eight-core machine you need eight processes, each with its own event loop, behind a load balancer or a shared socket -- which is exactly what uvicorn --workers, gunicorn with uvicorn workers, or N replicas in Kubernetes do. Adding threads inside the process does not multiply loop throughput; it only gives you somewhere to park blocking work.

As for the loop itself, uvloop remains the cheapest improvement available: a Cython implementation on top of libuv, a drop-in replacement, and on network-I/O-bound workloads it usually delivers a clear improvement without touching your code. Since 0.22 there are wheels for CPython 3.14, including the free-threaded variant. Measure it on your workload before you take it on faith: if your bottleneck is the database, the loop was never the problem.

How you install it has changed, and it is worth updating. The event loop policy system (asyncio.set_event_loop_policy) was deprecated in Python 3.15 and is scheduled for removal in 3.16. The replacement is asyncio.Runner with loop_factory, available since 3.11:

```
import asyncio

try:
    import uvloop
    loop_factory = uvloop.new_event_loop
except ImportError:               # without uvloop, the stdlib loop is fine
    loop_factory = asyncio.new_event_loop


def main() -> None:
    # asyncio.Runner (3.11+) replaces event loop policies, which were
    # deprecated in 3.15 and are slated for removal in 3.16.
    with asyncio.Runner(loop_factory=loop_factory) as runner:
        runner.run(server())


if __name__ == "__main__":
    main()
```

Worth mentioning too: asyncio.iscoroutinefunction was deprecated in 3.15 in favour of inspect.iscoroutinefunction. If you have infrastructure code that branches on whether something is a coroutine -- decorators, task dispatchers -- that change affects you.

And the structural news: 3.14 brings first-class free-threading support to asyncio, with a per-thread

linked list for native tasks that also delivered 10-20 % improvements on the standard benchmarks and lower memory use. On a free-threaded build you can run several event loops on different threads of the same process and scale roughly linearly. It is not the default deployment yet -- the C extension ecosystem is still catching up -- but it shifts the horizon: the question "one process per core, or threads?" no longer has a single obvious answer.

21. Worked example: a complete ingestion service

Let us put it all together. The service reads messages from a queue, calls an external API for each one, and writes the result to PostgreSQL. It has bounded concurrency, a time budget, clean shutdown and metrics. It is about sixty lines, and every decision comes from an earlier section.

```
import asyncio
import contextlib
import logging
import signal

import asyncpg
import httpx

log = logging.getLogger(__name__)
CONCURRENCY = 20


async def process(msg: dict, http: httpx.AsyncClient, pool: asyncpg.Pool) -> None:
    async with asyncio.timeout(5.0):                    # per-message budget
        r = await http.get(f"/enrich/{'{'}msg['id']{'}'}")
        r.raise_for_status()
        data = r.json()

    async with pool.acquire(timeout=2.0) as conn:       # never wait unbounded
        await conn.execute(
            "INSERT INTO events (id, payload) VALUES ($1, $2)"
            " ON CONFLICT (id) DO NOTHING",             # idempotent: safe to retry
            msg["id"], data,
        )


async def worker(n: int, queue: asyncio.Queue, http, pool) -> None:
    while True:
        msg = await queue.get()
        try:
            await process(msg, http, pool)
        except asyncio.CancelledError:
            await queue.put(msg)                        # put it back, do not lose it
            raise
        except Exception:
            log.exception("message %s failed", msg.get("id"))
        finally:
            queue.task_done()


async def main() -> None:
    stop = asyncio.Event()
    loop = asyncio.get_running_loop()
    for sig in (signal.SIGTERM, signal.SIGINT):
        loop.add_signal_handler(sig, stop.set)

    async with contextlib.AsyncExitStack() as stack:
        http = await stack.enter_async_context(
            httpx.AsyncClient(base_url=API, timeout=httpx.Timeout(4.0, connect=2.0))
        )
        pool = await stack.enter_async_context(
            asyncpg.create_pool(dsn=DSN, min_size=5, max_size=CONCURRENCY)
        )
        queue: asyncio.Queue = asyncio.Queue(maxsize=CONCURRENCY * 2)  # backpressure

        async with asyncio.TaskGroup() as tg:
```

```
            for n in range(CONCURRENCY):
                tg.create_task(worker(n, queue, http, pool), name=f"worker-{{}n{}}")
            producer = tg.create_task(feed(queue, stop), name="producer")

            await stop.wait()
            log.info("SIGTERM received; draining")
            producer.cancel()
            await queue.join()              # finish what is already in flight
            raise asyncio.CancelledError    # cancel the workers


if __name__ == "__main__":
    with contextlib.suppress(asyncio.CancelledError):
        asyncio.run(main())
```

The decisions that matter, in order. The queue maxsize is what stops a fast producer from filling memory; it is the same backpressure from section 6. The ON CONFLICT DO NOTHING makes the write idempotent, which is what lets you retry and requeue without duplicating. The await queue.put(msg) inside the CancelledError handler -- followed by raise -- is the difference between a shutdown that loses work and one that does not. And the await queue.join() before cancelling is what turns shutdown into draining: in-flight messages finish, new ones do not enter.

Pool size and concurrency are the same number on purpose. If you set CONCURRENCY = 100 against a pool of 20, you would have eighty workers waiting on a connection with their write timeout ticking: exactly the failure from section 12, now with the advantage that you would recognise it.

22. Runbook: the service is not responding

It is 3 a.m. and the service is alive but not answering. This is the sequence that gets you to a diagnosis in minutes instead of hours.

- Look at loop lag first. If the p99 is through the roof, someone is blocking the thread and the problem is CPU or a synchronous call. If lag is healthy and latency is high, the loop is free and you are waiting on something external: pool, network or a lock.
- Ask for the task tree. python -m asyncio pstree PID (3.14+) draws the call graph of every task across every thread, with no instrumentation and without stopping the process. If fifty tasks are waiting on the same line, you already have your answer. python -m asyncio ps PID gives the same information as a table, easier to filter.
- On builds without that tool, py-spy dump --pid PID gives you thread stacks. A main thread parked inside a function that is neither select nor epoll_wait is your blocker.
- Check pool metrics. A sustained in_use == max confirms exhaustion. Check slow queries on the database side at the same time: there is almost always one query that has degraded and is holding connections three times as long.
- Look for the lock. If the tree shows tasks waiting on Lock.acquire, someone is holding the lock across I/O. That is the pattern: a lock taken around a network call serializes the entire service.
- Mitigate, then fix. Restarting brings the service back, but if you did not capture the task tree first you have destroyed the evidence. A pstree dump costs one second and is the difference between fixing it today and waiting for the next time.

One tip that saves a lot of grief: paste that dump into the ticket. Asyncio hangs look very similar to each other on dashboards and very different in the task tree, which is where the information actually lives.

23. Frequently asked questions

Is asyncio faster than threads? For thousands of slow connections, yes, by a wide margin: a task costs kilobytes, a thread costs megabytes of stack. For a hundred connections and fast queries the difference is small and threads are easier to write. And for CPU-heavy work asyncio contributes nothing: it is still one thread.

Should I migrate my whole synchronous service to asyncio? Rarely. A partial migration is worse than none, because a single synchronous call on the hot path cancels the benefit of everything else. Migrate whole I/O-bound services, or do not migrate.

Which do I use, asyncio.timeout() or asyncio.wait_for()? asyncio.timeout(), from 3.11 onward. It is a context manager, it composes with nested timeouts, and it expresses a deadline over a block rather than over a single call. wait_for still works and is now implemented on top of it.

Can I use time.sleep() in a coroutine if it is really short? No. "Really short" multiplied by your request rate stops being short very quickly, and the cost is not paid by that request: it is paid by every other one waiting its turn on the loop.

Why is my exception missing from the logs? Almost certainly a fire-and-forget task whose result nobody collects. Use TaskGroup, or keep the reference and attach an add_done_callback that logs the failure. It is section 3, and it is the single most common bug.

How many concurrent tasks can I launch? The right question is how many your slowest dependency can absorb. The limit is not asyncio -- it handles tens of thousands -- but the database pool, the API rate limit, or the memory held by in-flight payloads. Start from that number and put a semaphore on it.

Is uvloop worth it? If your service is mostly network I/O and you are near the ceiling, yes, and it is a two-line change. If your p99 is dominated by a 300 ms query, the loop was not your problem.

Wrapping up

asyncio is not hard because of its syntax, which you can learn in an afternoon. It is hard because its execution model makes certain mistakes fail to fail -- they degrade instead. A service that blocks the loop raises no exceptions: it just gets slower. A task collected by the GC leaves no trace: it simply does not happen. A swallowed cancellation breaks nothing until the day a deploy takes ten minutes to drain.

Instrument the loop lag, bound concurrency at every fan-out, put timeouts everywhere with a budget that propagates, and treat cancellation as part of the contract of every coroutine you write. Those four habits cover the vast majority of the incidents you will ever see in an asyncio service.