PUIGFLOWS

Guía técnica · Proyectos Reales · Nivel intermedio

Continuous Profiling en Producción: Flame Graphs, pprof, py-spy, eBPF y la Cuarta Señal de OpenTelemetry

Edición bilingüe (español / English) - la señal de observabilidad que responde la pregunta que las trazas dejan a medias.

---

puigflows.com · Recurso premium
Actualizado: 16 de septiembre de 2026
Español: ~10962 palabras · ~50 min de lectura
English: ~10234 words · ~47 min read

VERSIÓN EN ESPAÑOL

Continuous Profiling en Producción: Flame Graphs, pprof, py-spy, eBPF y la Cuarta Señal de OpenTelemetry

El agujero que dejan las métricas, los logs y las trazas

Tienes un panel que dice que el p99 de /checkout pasó de 180 ms a 640 ms el martes por la tarde. Tienes trazas que señalan el span render_cart como responsable de casi todo ese tiempo. Tienes logs correlacionados por request_id. Y sigues sin saber qué tocar, porque ninguna de las tres señales te dice qué línea de código se está comiendo esos milisegundos.

Ese es el hueco exacto que cubre el profiling. Una métrica te dice cuánto. Una traza te dice dónde, con una resolución que llega hasta el span que alguien se molestó en instrumentar. Un perfil te dice qué: qué función, qué línea, qué pila de llamadas completa estaba consumiendo CPU o reservando memoria mientras el reloj corría. Es la única señal que no depende de que hayas anticipado el problema al instrumentar.

Durante años el profiling fue una actividad de laboratorio: reproducías el problema en tu portátil, lanzabas un profiler, mirabas un gráfico. Eso funciona para un algoritmo, y falla para casi todo lo demás. La carga de producción no se reproduce, los datos de producción no se reproducen y el hardware de producción tampoco. El continuous profiling -perfilar toda la flota, todo el rato, con una sobrecarga que puedes pagar- convierte esa actividad de laboratorio en una señal de observabilidad más, con histórico, con comparación entre versiones y con capacidad de responder preguntas después de que el incidente haya terminado.

Esta guía va de cómo montarlo sin romper nada: la estadística mínima que necesitas para no sacar conclusiones falsas, cómo leer un flame graph de verdad, la configuración correcta en Go, Python y Node.js, el salto a continuous profiling con Pyroscope y eBPF, y el estado real de la señal de perfiles de OpenTelemetry a día de hoy.

Muestreo frente a instrumentación: por qué un profiler de producción tiene que aproximar

Hay dos formas de construir un profiler. Un profiler determinista (o de trazado) intercepta cada entrada y salida de función y cronometra. Te da conteos de llamada exactos y una sobrecarga que va del 30% a más del 1000%, porque el coste de la instrumentación es proporcional al número de llamadas, no al tiempo de ejecución. cProfile en Python y --prof en V8 son de esta familia.

Un profiler de muestreo no intercepta nada. Cada N milisegundos interrumpe (o lee desde fuera) el proceso, captura la pila de llamadas y la guarda. El coste es proporcional a la frecuencia de muestreo y al número de hilos, no al trabajo que hace tu programa, y por eso es constante y predecible. Todos los profilers que puedes dejar corriendo en producción son de muestreo.

El precio es que el resultado es una estimación estadística. Merece la pena hacer las cuentas una vez, porque decide cuánto tiempo tienes que perfilar antes de creerte el resultado.

Con una frecuencia de 10 kHz durante 10 segundos recoges unas 100.000 muestras. Si una función aparece en 5.000 de ellas, estimas que consumió el 5% del tiempo, es decir unos 500 ms. El margen de error de esa cifra con 100.000 muestras es de aproximadamente ±0,5%. Con sólo 1.000 muestras -por ejemplo, 100 Hz durante 10 segundos- ese mismo 5% podría ser en realidad cualquier cosa entre el 3% y el 7%. La regla práctica: una diferencia por debajo del 2% no es señal, es ruido, salvo que hayas acumulado muchísimas muestras.

De ahí salen dos consecuencias que la gente aprende por las malas:

- No compares perfiles de 5 segundos. Para una comparación entre dos versiones, acumula minutos, no segundos. En continuous profiling esto sale gratis porque el backend agrega muestras de toda la flota y de toda la ventana temporal.
- Las funciones muy cortas desaparecen. Una función que dura 20 µs y se llama mil veces por segundo sí aparecerá (20 ms/s de CPU acumulados). Una que dura 20 µs y se llama tres veces al día no aparecerá nunca. El muestreo mide tiempo agregado, no eventos.

Un detalle pequeño que sí importa: elige frecuencias primas o impares, como 99 Hz o 97 Hz, en lugar de 100 Hz. Muchos sistemas tienen trabajo periódico anclado a múltiplos de 10 ms (timers del kernel, tickers de la aplicación, GC pacing). Si muestreas exactamente a 100 Hz corres el riesgo de sincronizarte con ese trabajo y sobrerrepresentarlo o no verlo nunca. A 99 Hz la fase deriva y el sesgo desaparece. Es la razón por la que el ejemplo canónico de perf record usa -F 99.

Cómo leer un flame graph sin engañarte

El flame graph es la visualización estándar de un perfil, y también la más malinterpretada. Tres reglas bastan para leerlo bien:

1. El eje X no es tiempo. Esto es lo que casi todo el mundo asume y es falso. Las cajas se ordenan alfabéticamente dentro de cada nivel para que las pilas idénticas se fusionen. Un flame graph no tiene una línea temporal: si quieres ver la secuencia de eventos, necesitas una traza de ejecución, no un perfil.
2. La anchura es número de muestras, es decir tiempo acumulado. Una caja que ocupa el 30% del ancho significa que esa pila estaba activa en el 30% de las muestras. Nada más y nada menos.
3. Las mesetas planas son el objetivo. Una caja ancha con hijos anchos sólo significa que llama a algo caro; el trabajo real está más abajo. Una caja ancha sin hijos, una meseta en la parte alta del gráfico, es self time: ahí se está quemando CPU de verdad. Cuando alguien dice "busca las mesetas", esto es lo que quiere decir.

El flame graph diferencial es la herramienta que convierte el profiling en algo operativo. Tomas un perfil de la versión anterior como base, otro de la nueva, y el gráfico colorea en rojo lo que creció y en azul lo que encogió. Es cómo se localiza una regresión de rendimiento en minutos en lugar de en días. La trampa: si las dos ventanas tienen cargas distintas, todo crece o todo encoge y el diferencial no significa nada. Normaliza siempre por una unidad de trabajo comparable (CPU por petición, no CPU total) o compara ventanas con tráfico equivalente.

Por último, el icicle graph (carámbanos) es el mismo gráfico dibujado hacia abajo desde la raíz. Pyroscope y el visor web de pprof usan esta orientación por defecto. No cambia nada semánticamente; sólo que la raíz está arriba.

Go: pprof bien puesto

Go trae el mejor profiler integrado de cualquier lenguaje mainstream, y también la forma más fácil de exponer tu proceso a internet sin darte cuenta. Empecemos por ahí.

El truco que todo el mundo copia es import _ "net/http/pprof". Ese import en blanco tiene un efecto secundario: registra los handlers de pprof en http.DefaultServeMux. Si tu servidor público también usa el DefaultServeMux -y lo hace si llamas a http.ListenAndServe(":8080", nil)- acabas de publicar /debug/pprof/heap en internet. Eso es un volcado de memoria con nombres de funciones, y /debug/pprof/profile?seconds=3600 es además una negación de servicio gratuita.

La forma correcta es importar el paquete con nombre y registrar los handlers en un mux de administración propio, escuchando en loopback o en un puerto que no publiques:

```
package main

import (
"log"
"net/http"
"net/http/pprof"
"runtime"
"time"
)

func main() {
// Contención de mutex: se registra 1 de cada 5 eventos. 0 lo desactiva.
runtime.SetMutexProfileFraction(5)

// Bloqueo: el runtime aspira a una muestra por cada 10.000 ns
// (10 µs) de tiempo bloqueado. 0 lo desactiva.
runtime.SetBlockProfileRate(10_000)

// Heap: una muestra por cada 512 KiB asignados. Es el valor por
// defecto; bajarlo da más detalle y cuesta más CPU.
runtime.MemProfileRate = 512 * 1024

// pprof vive en su propio mux, nunca en el público.
admin := http.NewServeMux()
admin.HandleFunc("/debug/pprof/", pprof.Index)
admin.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
admin.HandleFunc("/debug/pprof/profile", pprof.Profile)
admin.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
admin.HandleFunc("/debug/pprof/trace", pprof.Trace)

go func() {
srv := &http.Server{
Addr: "127.0.0.1:6060",
Handler: admin,
ReadTimeout: 5 * time.Second,
// Un perfil de CPU de 60 s no puede morir por timeout de escritura.
WriteTimeout: 150 * time.Second,
}
log.Fatal(srv.ListenAndServe())
}()

// ... aquí tu servidor público, en otro mux y otro puerto.
}
```

Ese WriteTimeout es el error silencioso número uno: pides ?seconds=60, el servidor corta la conexión a los 30 y te quedas mirando un perfil truncado sin ningún mensaje que lo explique.

Los seis perfiles y a qué pregunta responde cada uno

- profile - CPU. Muestreado a 100 Hz por goroutine en ejecución. Responde: ¿en qué se quema el tiempo de CPU? Es el que quieres el 80% de las veces.
- heap - memoria. Responde dos preguntas distintas según el índice que mires: inuse_space es lo que sigue vivo ahora mismo (para fugas), alloc_space es todo lo asignado desde el arranque (para presión de GC). Confundirlos es el clásico de la sección siguiente.
- goroutine - cuántas goroutines hay y dónde están paradas. Es lo primero que se mira ante una fuga de goroutines.
- mutex - dónde se pierde tiempo esperando un sync.Mutex. Desactivado por defecto.
- block - dónde se bloquean las goroutines: canales, select, WaitGroup, syscalls. Desactivado por defecto.
- allocs - alias de heap con el índice por defecto en alloc_space.

El consejo operativo: deja mutex y block activados de forma permanente en tu profiler continuo, con fracciones conservadoras como las del ejemplo. Una proporción grande de las regresiones de rendimiento no son de CPU sino de contención, y son justo las que el perfil de CPU no muestra: una goroutine esperando un lock no consume CPU, así que no aparece en ninguna muestra.

El flujo de trabajo con la herramienta

```
# Perfil de CPU de 30 segundos, abierto en el visor web interactivo.
go tool pprof -http=:8080 http://127.0.0.1:6060/debug/pprof/profile?seconds=30

# Memoria viva ahora mismo (fugas).
go tool pprof -http=:8080 -sample_index=inuse_space \
http://127.0.0.1:6060/debug/pprof/heap

# Volumen total asignado (presión de GC).
go tool pprof -http=:8080 -sample_index=alloc_space \
http://127.0.0.1:6060/debug/pprof/heap

# Guardar una línea base y comparar contra ella tras un cambio.
curl -o base.pprof "http://127.0.0.1:6060/debug/pprof/profile?seconds=60"
# ... despliegas el cambio ...
curl -o new.pprof "http://127.0.0.1:6060/debug/pprof/profile?seconds=60"
go tool pprof -http=:8080 -base base.pprof new.pprof
```

Dentro del visor, las vistas que se usan de verdad son Flame Graph para orientarte, Top ordenado por flat (self time) para la lista de sospechosos, y Source para ver el desglose línea a línea de una función concreta. El modo -base es el flame graph diferencial: rojo lo que creció, azul lo que encogió.

Etiquetas: atribuir CPU por tenant, por ruta o por versión

Un perfil agregado de un servicio multi-tenant te dice que el 40% de la CPU se va en deserializar JSON. Útil a medias. Lo que quieres saber es si ese 40% viene repartido entre mil clientes o si hay uno que está haciendo peticiones de 30 MB. Las etiquetas de pprof resuelven eso:

```
import (
"context"
"net/http"
"runtime/pprof"
)
```

```
func profileLabels(next http.Handler) http.Handler {
return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
labels := pprof.Labels(
"tenant", tenantFromRequest(r),
"route", routePattern(r), // el patrón, no la URL con IDs
)
pprof.Do(r.Context(), labels, func(ctx context.Context) {
next.ServeHTTP(w, r.WithContext(ctx))
})
})
}
```

Dos detalles que cuestan una tarde si no los sabes. Primero: sólo el perfil de CPU registra etiquetas; el de heap no las lleva, así que no puedes atribuir memoria por tenant de esta forma. Segundo: las etiquetas se heredan en las goroutines que lances después de aplicarlas dentro del mismo contexto, pero no se aplican retroactivamente a un worker pool que ya estaba corriendo. Si haces fan-out a un pool de workers preexistente, el trabajo pesado aparecerá sin etiqueta.

Y una regla de higiene: usa el patrón de ruta (/users/{id}), no la URL concreta. Igual que en Prometheus, una etiqueta con IDs dentro te revienta la cardinalidad del backend de perfiles.

alloc_space frente a inuse_space, el error que manda a optimizar la función equivocada

Un servicio consume 4 GB y crees que tienes una fuga. Abres el perfil de heap, ves que json.Unmarshal aparece con 300 GB y te lanzas a optimizar la deserialización. Pero estabas mirando alloc_space, que es el acumulado desde el arranque del proceso: 300 GB repartidos en seis días de uptime son 2 GB por hora, perfectamente normales y todos ellos recolectados por el GC. La fuga está en otro sitio, y sólo la ve inuse_space.

La forma corta de recordarlo: inuse_space responde "¿quién retiene memoria?" y es para fugas; alloc_space responde "¿quién genera basura?" y es para presión de GC y latencia. Ambas preguntas son legítimas; contestarlas con el índice equivocado no.

Cuando el problema no es CPU: el FlightRecorder de Go 1.25

Hay una clase de problema que ningún perfil de CPU puede explicar: la petición que tardó 800 ms porque durante 700 ms no se estaba ejecutando nada. Esperaba un lock, una syscall, un canal o al recolector de basura. Para eso están las trazas de ejecución, pero activar el tracer en un servicio de larga vida genera megabytes por segundo de datos que nadie va a mirar.

Go 1.25 resolvió esto con runtime/trace.FlightRecorder: el tracer corre siempre, pero en lugar de escribir a disco mantiene en memoria una ventana circular con los últimos segundos. Cuando tu código detecta la anomalía, vuelca esa ventana. Es exactamente la caja negra de un avión.

```
import (
"log"
"os"
"runtime/trace"
"sync"
"time"
)

var once sync.Once
```

```
func main() {
fr := trace.NewFlightRecorder(trace.FlightRecorderConfig{
// Ventana retenida de forma fiable. La guía oficial recomienda
// ~2x el tamaño del evento que investigas: para un timeout de
// 5 s, pon 10 s.
MinAge: 2 * time.Second,
// Tope de memoria. Un servicio ocupado genera del orden de
// 10 MB/s de traza, así que este límite es el que manda.
MaxBytes: 16 << 20, // 16 MiB
})
if err := fr.Start(); err != nil {
log.Printf("flight recorder no disponible: %v", err)
}
defer fr.Stop()

// En el handler, tras medir la latencia:
// if fr.Enabled() && time.Since(start) > 500*time.Millisecond {
// go captureSnapshot(fr)
// }
}

func captureSnapshot(fr *trace.FlightRecorder) {
// once evita que una tormenta de peticiones lentas escriba
// mil ficheros de traza.
once.Do(func() {
f, err := os.Create("/tmp/snapshot.trace")
if err != nil {
log.Printf("no se pudo crear el snapshot: %v", err)
return
}
defer f.Close()
if _, err := fr.WriteTo(f); err != nil {
log.Printf("no se pudo escribir el snapshot: %v", err)
return
}
log.Printf("snapshot de traza capturado en %s", f.Name())
})
}
```

Después se abre con go tool trace /tmp/snapshot.trace. La vista "View trace by proc" muestra huecos: momentos en los que todos los procesadores están parados. Esos huecos son la respuesta que el perfil de CPU nunca te iba a dar.

Perfil y traza no compiten: el perfil responde "dónde se gastó la CPU", la traza responde "por qué no se estaba gastando".

Python: py-spy hoy, Tachyon mañana

En Python la situación es distinta porque el profiler de la biblioteca estándar, cProfile, no sirve para producción. Es determinista: instrumenta cada llamada. Además de multiplicar por dos o por diez el tiempo de ejecución, introduce un sesgo sistemático, porque el coste de la instrumentación es proporcional al número de llamadas. Un código con muchas funciones pequeñas aparece mucho más caro de lo que es, y acabas "optimizando" algo que en producción no cuesta nada. Usa cProfile en desarrollo y para scripts cortos; para producción, un profiler de muestreo.

py-spy: adjuntarse a un proceso sin tocarlo

py-spy es el estándar de facto. Está escrito en Rust, corre fuera del proceso objetivo -lee su memoria directamente- y no requiere ni un solo cambio en tu código ni reiniciar nada. Es la herramienta que usas

a las tres de la mañana cuando un worker lleva veinte minutos al 100% de CPU y nadie sabe por qué.

```
# Vista tipo "top", en vivo, de las funciones más calientes.
py-spy top --pid 4242

# Grabar 60 segundos y generar un flame graph SVG.
py-spy record --pid 4242 --duration 60 --output flame.svg

# Un worker de Gunicorn/Celery con procesos hijo: sigue el árbol entero.
py-spy record --pid 4242 --duration 60 --subprocesses --output flame.svg

# Sólo el tiempo en el que el hilo realmente sostiene la GIL. La diferencia
# entre esto y el perfil normal es tu tiempo de espera de E/S.
py-spy record --pid 4242 --duration 60 --gil --output gil.svg

# Instantánea única de todas las pilas. Esto es lo que lanzas contra un
# proceso colgado: te dice en qué línea exacta está clavado.
py-spy dump --pid 4242
```

El obstáculo real es de permisos, no de la herramienta. py-spy lee la memoria de otro proceso y eso está restringido:

```
# Linux: el ptrace_scope de Yama vale 1 por defecto, lo que sólo permite
# adjuntarse a procesos hijo. Si ves "Operation not permitted", es esto.
cat /proc/sys/kernel/yama/ptrace_scope
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope # temporal, hasta reiniciar

# Docker: el contenedor necesita la capacidad explícitamente.
docker run --cap-add SYS_PTRACE myapp
```

En Kubernetes el patrón que funciona es un contenedor de diagnóstico que comparte el espacio de nombres de procesos con la aplicación, de modo que la aplicación no necesita privilegios extra:

```
apiVersion: v1
kind: Pod
metadata:
name: checkout-api
spec:
shareProcessNamespace: true # el sidecar ve los PIDs de la app
containers:
- name: app
image: registry.example.com/checkout-api:1.4.2
- name: profiler
image: ghcr.io/benfred/py-spy:latest
command: ["sleep", "infinity"]
securityContext:
capabilities:
add: ["SYS_PTRACE"]
runAsUser: 0
resources:
limits:
cpu: 200m
memory: 128Mi
```

Para memoria, py-spy no sirve: usa memray, que rastrea cada asignación, incluidas las de extensiones nativas en C, que es justo donde tracemalloc se queda ciego.

```
memray run --native -o salida.bin -m miapp.worker
memray flamegraph salida.bin # genera un HTML navegable
memray attach 4242 # adjuntarse a un proceso ya en marcha
```

Lo que llega en Python 3.15: profiling.sampling (Tachyon)

Python 3.15 incorpora por fin un profiler de muestreo a la biblioteca estándar. La PEP 799 reorganiza los profilers en un paquete profiling con dos módulos: profiling.tracing (el determinista de siempre) y profiling.sampling, cuyo nombre en clave es Tachyon. Se apoya en la interfaz de adjuntado seguro que introdujo la PEP 768, así que igual que py-spy lee el proceso desde fuera y, según la documentación, no impone sobrecarga medible sobre el objetivo.

```
# Adjuntarse a un proceso vivo 30 s y escribir un flame graph autocontenido.
python -m profiling.sampling attach --flamegraph -d 30 -o perfil.html 12345

# Sólo el tiempo con la GIL sostenida (hay modos wall, cpu, gil y exception).
python -m profiling.sampling attach --mode=gil -d 30 12345

# Instantánea única de la pila de todos los hilos: proceso colgado.
python -m profiling.sampling dump --all-threads 12345

# Aplicaciones asyncio: reconstruye la pila a través de los await.
python -m profiling.sampling run --async-aware --async-mode=all worker.py

# Guardar una base binaria y comparar después con un flamegraph diferencial.
python -m profiling.sampling run --binary -o base.bin -m miapp
python -m profiling.sampling run --diff-flamegraph base.bin -o diff.html -m miapp
```

La frecuencia por defecto es 1 kHz y se ajusta con -r (-r 10khz). Tres límites que conviene conocer antes de planificar nada con esto:

- El profiler y el proceso objetivo tienen que ejecutar la misma versión menor de Python. No puedes perfilar un proceso 3.15 desde un intérprete 3.14. Con versiones preliminares la exigencia es aún mayor: la versión exacta. Y no se puede cruzar entre un build free-threaded y uno estándar.
- Los requisitos de permisos son los mismos que los de py-spy: ptrace/process_vm_readv en Linux con CAP_SYS_PTRACE o ptrace_scope a 0; task_for_pid() en macOS; privilegios de administrador o SeDebugPrivilege en Windows.
- El modo --async-aware es incompatible con --native, --all-threads y con los modos cpu y gil, porque usa un mecanismo distinto de reconstrucción de pila.

Mientras tanto, py-spy sigue siendo la opción sensata: funciona con todo lo que ya tienes desplegado, incluidas las versiones de Python que llevan años en producción.

Node.js: --cpu-prof y una sesión de inspector

En Node el camino rápido es una bandera de arranque. Al terminar el proceso, V8 escribe un fichero .cpuprofile que se abre arrastrándolo a la pestaña Performance de Chrome DevTools o a speedscope:

```
node --cpu-prof --cpu-prof-dir=/tmp/perfiles servidor.js
```

Eso sirve para un script, no para un servicio que lleva nueve días arriba y al que no vas a reiniciar. Para eso se abre una sesión del inspector desde dentro del propio proceso, disparada por una señal o por un endpoint de administración:

```
const inspector = require('node:inspector');
const fs = require('node:fs/promises');
const path = require('node:path');
```

```
let activa = false;

async function capturarPerfilCPU(segundos = 30) {
if (activa) throw new Error('ya hay una captura en curso');
activa = true;

const session = new inspector.Session();
session.connect();

const post = (metodo, params) =>
new Promise((resolve, reject) =>
session.post(metodo, params, (err, res) =>
err ? reject(err) : resolve(res)));

try {
await post('Profiler.enable');
// Intervalo de muestreo en microsegundos. 1000 µs = 1 kHz.
await post('Profiler.setSamplingInterval', { interval: 1000 });
await post('Profiler.start');

await new Promise((r) => setTimeout(r, segundos * 1000));

const { profile } = await post('Profiler.stop');
const destino = path.join('/tmp', 'cpu-' + Date.now() + '.cpuprofile');
await fs.writeFile(destino, JSON.stringify(profile));
return destino;
} finally {
session.disconnect();
activa = false;
}
}

// Disparo por señal: kill -USR2 PID y listo, sin reiniciar nada.
process.on('SIGUSR2', () => {
capturarPerfilCPU(30)
.then((f) => console.log({ msg: 'perfil escrito', file: f }))
.catch((e) => console.error({ msg: 'perfil fallido', err: e.message }));
});
```

La guarda activa no es decorativa: dos capturas simultáneas de V8 en el mismo proceso se pisan y el resultado es basura.

Para memoria, require('node:v8').writeHeapSnapshot() existe y funciona, pero con una advertencia grande: detiene el mundo durante toda la captura y el fichero es del tamaño del heap. En un proceso con 3 GB de heap eso son varios segundos de pausa total y 3 GB de disco. Es una herramienta de última instancia, nunca algo que programes cada cinco minutos.

De perfilar a mano a continuous profiling

Todo lo anterior comparte un defecto: hace falta que estés delante. El incidente ocurre a las 03:40, el pod se reinicia a las 03:47 y a las 09:00, cuando abres el portátil, el proceso que querías perfilar ya no existe. El continuous profiling resuelve eso de la única forma posible: perfilando siempre, guardando las muestras con etiquetas y dejándote consultar hacia atrás.

Las cifras que hacen esto viable son modestas. Un agente de muestreo típico, como Grafana Pyroscope, añade del orden de un 2-5% de CPU y menos de 50 MB de memoria por pod, incluso con varios tipos de perfil activados a la vez. A cambio obtienes, por cada despliegue, la capacidad de preguntar "¿qué función consume más CPU por petición ahora que antes?" y recibir una respuesta en treinta segundos.

Dos formas de recoger: SDK o eBPF

Con SDK instrumentas la aplicación. A cambio de un import y unas líneas de configuración obtienes símbolos perfectos, etiquetas de negocio y correlación con trazas. En Python:

```
import os
import pyroscope

pyroscope.configure(
application_name="checkout-api",
server_address=os.environ["PYROSCOPE_URL"],
# Sólo contabiliza tiempo de CPU; con False mide reloj de pared
# (útil si tu cuello de botella es E/S y no cálculo).
oncpu=True,
# Estas etiquetas son lo que convierte el perfil en algo accionable.
tags={
"env": os.environ["ENVIRONMENT"],
"region": os.environ["AWS_REGION"],
# El SHA del despliegue: sin esto no puedes atribuir una
# regresión a un cambio concreto.
"version": os.environ["GIT_SHA"],
},
)

# Etiquetado dinámico por unidad de trabajo, dentro de un worker.
def procesar(job):
with pyroscope.tag_wrapper({"job_type": job.kind, "tenant": job.tenant_id}):
return ejecutar(job)
```

La etiqueta version es la que más rendimiento da por línea de código escrita. Con ella, "el p99 subió tras el despliegue de ayer" deja de ser una hipótesis: filtras el perfil por los dos SHA, pides el diferencial y ves la función que engordó.

Con eBPF no tocas la aplicación. Un agente privilegiado por nodo muestrea las pilas de todos los procesos desde el kernel. Es la opción para parques heterogéneos, binarios de terceros y equipos que no van a añadir una dependencia a cuarenta servicios. La configuración con Grafana Alloy como DaemonSet:

```
discovery.kubernetes "pods" {
role = "pod"
}

discovery.relabel "pods_locales" {
targets = discovery.kubernetes.pods.targets

// Cada agente perfila sólo los pods de SU nodo. Sin esta regla,
// cada agente intenta perfilar el clúster entero.
rule {
source_labels = ["__meta_kubernetes_pod_node_name"]
regex = env("HOSTNAME_NODE")
action = "keep"
}

rule {
source_labels = ["__meta_kubernetes_namespace", "__meta_kubernetes_pod_label_app"]
separator = "/"
target_label = "service_name"
}
}

pyroscope.ebpf "default" {
forward_to = [pyroscope.write.backend.receiver]
targets = discovery.relabel.pods_locales.output
```

```
sample_rate = 97 // impar, para no sincronizar con timers
collect_kernel_profile = false
}

pyroscope.write "backend" {
endpoint {
url = "http://pyroscope.observability.svc:4040"
}
external_labels = {
cluster = "prod-eu-west-1",
}
}
```

El precio de eBPF son dos limitaciones concretas que conviene conocer antes de elegir este camino:

- Frame pointers. Para reconstruir la pila desde el kernel hace falta que los binarios se hayan compilado con punteros de marco. El desenrollado con DWARF no está disponible dentro del kernel, así que si el binario y sus bibliotecas se compilaron con -fomit-frame-pointer -el comportamiento por defecto durante dos décadas- obtienes pilas truncadas o directamente inútiles. La buena noticia es que la industria dio marcha atrás: Fedora los reactivó por defecto en la versión 38 y Ubuntu hizo lo propio en 24.04 LTS para plataformas de 64 bits. Los binarios de Go los llevan siempre. Si tu base es una distro antigua o compilas C/C++ tú mismo, añade -fno-omit-frame-pointer a tus flags o el perfil no valdrá nada.
- Lenguajes interpretados y con JIT. Un perfil de eBPF sobre un proceso de Python sin unwinder específico te muestra _PyEval_EvalFrameDefault repetido cuarenta veces: cierto, e inútil. Necesitas que el agente tenga un desenrollador que sepa leer las estructuras del intérprete. Los agentes modernos lo traen para Python, Java, Ruby, PHP, .NET, Node.js y Perl, además de soporte nativo para C/C++, Go, Rust y Zig. Comprueba la matriz de tu agente contra tu parque antes de prometer cobertura total.

El criterio práctico: eBPF para cobertura amplia y barata, SDK para los servicios que de verdad te importan, donde quieres etiquetas de negocio y enlace con trazas. No son excluyentes.

Cardinalidad: el mismo error que ya cometiste con Prometheus

Cada combinación única de etiquetas es una serie de perfiles separada que el backend debe almacenar e indexar. Si añades user_id como etiqueta con cien mil usuarios, has multiplicado tu coste de almacenamiento por cien mil. Las reglas son idénticas a las de las métricas: etiquetas de baja cardinalidad y acotadas (service, env, region, version, route_pattern), nunca identificadores libres. El SHA de la versión es la excepción tolerable, porque su cardinalidad crece con el ritmo de despliegue y se puede recortar con retención.

La cuarta señal: OpenTelemetry Profiles

Hasta ahora, cada backend de perfiles hablaba su propio idioma y cambiar de proveedor significaba cambiar de agente en toda la flota. OpenTelemetry está cerrando ese hueco: la señal de perfiles entró en alpha pública en marzo de 2026, convirtiendo el profiling en el cuarto pilar junto a trazas, métricas y logs.

Dos decisiones de diseño del formato OTLP de perfiles merecen mención. La primera es que hace round-trip sin pérdidas con pprof, el formato de facto desde hace una década: puedes convertir en ambos sentidos sin perder información, lo que significa que todo tu utillaje existente sigue sirviendo. La

segunda es un diccionario de cadenas compartido que reduce el tamaño en el cable en torno a un 40%, algo nada trivial cuando multiplicas por miles de pods.

La advertencia honesta: alpha significa alpha. La especificación OTLP es estable para trazas, métricas y logs, pero sigue en desarrollo para perfiles, y el propio SIG desaconseja apoyar cargas críticas sobre ella todavía. Lo razonable en 2026 es tratarla como ventana de evaluación: monta tu continuous profiling con lo que funciona hoy (Pyroscope, Parca, o el producto de tu proveedor) y ve probando la señal OTel en un entorno no crítico para que la migración, cuando llegue beta y GA, sea un cambio de exportador y no un proyecto.

Enlazar trazas y perfiles hoy: span profiles

Lo que sí puedes tener ya es el salto directo desde un span lento a las pilas que se ejecutaron durante ese span. El mecanismo es sencillo: cuando un span arranca, se etiquetan las muestras de perfil con su span_id. Con eso, la interfaz de trazas puede mostrar un flame graph de ese span concreto.

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from pyroscope.otel import PyroscopeSpanProcessor

provider = TracerProvider()
provider.add_span_processor(BatchSpanProcessor(OTLPSpanExporter()))
# Etiqueta las muestras de perfil con el span activo.
provider.add_span_processor(PyroscopeSpanProcessor())
trace.set_tracer_provider(provider)
```

El resultado operativo es el que abre esta guía, pero resuelto: ves un span render_cart de 640 ms, pulsas en la pestaña de perfil y encuentras que el 70% se va en una regex compilada dentro de un bucle. Eso es la diferencia entre saber que algo va lento y saber qué arreglar.

Memoria: cuando el heap profile no dice nada

Hay un escenario que desconcierta a mucha gente. El pod se muere por OOMKilled con un límite de 2 GiB. Abres el perfil de heap y suma 900 MB. Los números no cuadran y la conclusión precipitada es que el profiler miente. No miente: está midiendo otra cosa.

La RSS de un proceso es el heap del runtime más varias cosas que ningún perfil de heap gestionado va a mostrarte:

- Fragmentación y memoria devuelta con pereza. El asignador reserva del sistema operativo en bloques grandes y no siempre los devuelve cuando el objeto muere. En Go, runtime/metrics y go_memstats_heap_released_bytes te dan la foto real; en glibc, la cantidad de arenas por hilo puede inflar la RSS de forma espectacular en procesos muy multihilo, y MALLOC_ARENA_MAX=2 es un arreglo clásico.
- Asignaciones nativas. Un búfer reservado dentro de una extensión en C -NumPy, un cliente de base de datos, un códec de imagen- no pasa por el asignador del lenguaje y no aparece en el perfil. En Python esto es exactamente lo que separa a tracemalloc (sólo ve asignaciones del intérprete) de memray --native (las ve todas).
- Pilas de hilos, mapeos de ficheros, el propio binario y sus bibliotecas. Diez mil goroutines con 8 KiB de pila inicial son 80 MiB que no están en el heap profile.

La secuencia de diagnóstico que funciona: primero compara RSS (desde container_memory_working_set_bytes, no desde free) contra el heap vivo que reporta el runtime. Si el heap crece, tienes una fuga de objetos y el perfil de inuse_space te la señala. Si el heap es plano y la RSS sube, el problema es fragmentación o asignaciones nativas, y ahí necesitas un profiler que baje al nivel del asignador del sistema.

En Go hay además una palanca que resuelve muchos OOMKilled sin cambiar una línea de lógica: GOMEMLIMIT. Fijarlo en torno al 90% del límite del contenedor hace que el recolector trabaje más agresivamente al acercarse al techo en lugar de dejar que el kernel mate el proceso. No arregla una fuga real, pero convierte un pico transitorio en un poco más de CPU en vez de en una caída.

La JVM: JFR y async-profiler, o por qué tu flame graph de Java mentía

Si tu plataforma es Java o Kotlin, el capítulo de Go no te sirve tal cual, y no por sintaxis: los profilers de la JVM tienen un problema que los de Go y Python no tienen. La mayoría de las herramientas clásicas -VisualVM, JProfiler en modo muestreo, cualquier cosa construida sobre ThreadMXBean.getThreadInfo()- sólo pueden capturar la pila de un hilo cuando ese hilo está en un safepoint. Y los safepoints no están repartidos de forma uniforme por el código: el JIT los coloca en los saltos hacia atrás de los bucles, en las llamadas a métodos y en las asignaciones, pero los elimina agresivamente de los bucles contados sobre arrays y del código intrínseco. El resultado tiene nombre propio: safepoint bias. El profiler no te enseña dónde está la CPU, te enseña dónde están los safepoints más cercanos a donde está la CPU. Un bucle numérico que consume el 40% del proceso puede aparecer atribuido al método que lo llama, o no aparecer en absoluto.

La consecuencia práctica es incómoda: mucha gente ha optimizado la función equivocada durante años con datos que parecían correctos porque el flame graph se dibujaba bien. Un flame graph bonito no es un flame graph cierto.

async-profiler: muestreo sin safepoint bias

async-profiler resuelve el problema usando AsyncGetCallTrace, una API interna de HotSpot que puede desenrollar la pila de un hilo Java desde un manejador de señal, sin esperar a un safepoint. La señal la entrega el subsistema de eventos de rendimiento del kernel (perf_events) o un temporizador POSIX, así que el muestreo es realmente proporcional al tiempo de CPU consumido. Además desenrolla la pila nativa, de modo que ves el código JIT, el intérprete, las llamadas JNI y el propio runtime de la JVM en el mismo gráfico: cuando el 30% del tiempo se va en G1ParScanThreadState o en jbyte_disjoint_arraycopy, lo ves, y ningún profiler de sólo-Java te lo habría enseñado.

La versión 4.3, publicada en enero de 2026, es la primera del año y consolidó dos cosas que importan en producción: la emisión nativa en formato JFR y la mejora del muestreo de reloj de pared. El proyecto ha adoptado un ritmo de una versión cada tres meses, lo que en la práctica significa que puedes fijar una versión y actualizar por trimestres en lugar de perseguir master.

```
# Adjuntarse a un proceso en marcha por PID, 30 segundos de CPU, salida HTML
./asprof -d 30 -e cpu -f /tmp/perfil-cpu.html $(pgrep -f mi-servicio.jar)

# Reloj de pared: incluye el tiempo bloqueado, no solo el tiempo en CPU.
# Imprescindible cuando la latencia sube pero la CPU no.
./asprof -d 30 -e wall -t -f /tmp/perfil-wall.html <pid>
```

```
# Contencion de locks y asignaciones, los dos perfiles que mas problemas
# resuelven en servicios Java despues del de CPU
./asprof -d 30 -e lock -f /tmp/perfil-lock.jfr <pid>
./asprof -d 30 -e alloc -f /tmp/perfil-alloc.jfr <pid>

# Como agente, desde el arranque, grabando en JFR de forma continua.
# --wall anade muestreo de reloj de pared a su propio intervalo, mas grueso,
# asi que una sola grabacion contiene on-CPU y off-CPU a la vez.
java
-agentpath:/opt/async-profiler/lib/libasyncProfiler.so=start,event=cpu,--wall=50ms,jfr,file=/var/log/
perf-%p-%t.jfr,loop=30m \
-jar mi-servicio.jar
```

Dos detalles operativos que cuestan una tarde si no los sabes. El primero: el desenrollado de pila necesita que la JVM arranque con -XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints, porque sin eso el mapa de depuración del JIT sólo tiene entradas en los safepoints y vuelves, por otra vía, al problema que querías evitar. El segundo: dentro de un contenedor, adjuntarse a un proceso requiere el mismo espacio de nombres de PID y la capacidad SYS_PTRACE, exactamente igual que con py-spy; si tu plan es perfilar bajo demanda en Kubernetes, o incluyes el agente en la imagen o usas un contenedor efímero con shareProcessNamespace: true.

JFR: el que ya tienes instalado y casi nadie enciende

JDK Flight Recorder lleva en el OpenJDK desde Java 11 (y retroportado a 8u), está en el propio JDK y su configuración default está diseñada desde los tiempos de JRockit para mantenerse por debajo del 1% de sobrecarga. Ese objetivo se ha sostenido versión tras versión: en el conjunto de referencia SPECjbb2015 la configuración por defecto se mide consistentemente en torno al 1% o menos. Son más de 500 tipos de evento -GC, JIT, hilos, sockets, E/S de ficheros, excepciones, carga de clases, estadísticas de heap- y, a diferencia de un profiler externo, no tienes que instalar nada ni adjuntarte a nada.

Lo que históricamente le faltaba a JFR era precisamente un perfil de CPU honesto: su evento ExecutionSample arrastraba el mismo sesgo de safepoints. Eso cambió en el ciclo de JDK 25 LTS, donde JEP 509 introdujo el muestreo de CPU de JFR basado en el reloj de CPU del kernel en Linux, es decir, el mismo enfoque conceptual que async-profiler pero dentro del JDK y sin agentes de terceros. Si estás en JDK 25 o posterior en Linux, el argumento de «no puedo instalar un .so en producción» desapareció.

```
# Grabacion continua en anillo: siempre tienes los ultimos 15 minutos en memoria
# y un volcado a disco cuando algo va mal. Coste < 1%.
java -XX:StartFlightRecording=settings=default,maxage=15m,maxsize=200m,name=cont \
-XX:FlightRecorderOptions=stackdepth=128 \
-jar mi-servicio.jar

# Volcar el anillo AHORA, sin reiniciar nada, cuando salta la alerta
jcmd $(pgrep -f mi-servicio.jar) JFR.dump name=cont filename=/tmp/incidente.jfr

# Leer sin abrir JMC: el JDK trae un lector de linea de comandos
jfr summary /tmp/incidente.jfr
jfr print --events ExecutionSample --stack-depth 16 /tmp/incidente.jfr | head -60
jfr print --events JavaMonitorEnter,ThreadPark /tmp/incidente.jfr | head -40
```

La combinación que mejor funciona en la práctica no es elegir entre los dos, sino usar cada uno para lo que es bueno: JFR encendido siempre, en modo anillo, como caja negra del proceso -te da el contexto completo del minuto anterior al incidente, incluidas pausas de GC y eventos de E/S que ningún profiler

de muestreo registra-, y async-profiler bajo demanda cuando necesitas un flame graph de CPU o de asignaciones con resolución fina. Un detalle de versiones: el campo timeSpan que 4.3 añadió al evento profiler.WallClockSample hace que los conversores de JFR antiguos fallen al procesar grabaciones nuevas; el conversor nuevo sí lee las antiguas, así que la regla es actualizar primero la herramienta de lectura y después el agente.Off-CPU: el perfil que explica los 800 ms que la CPU no gastó

Off-CPU: el perfil que explica los 800 ms que la CPU no gastó

Hay una clase entera de problemas de latencia ante los que el perfil de CPU es ciego por construcción, y es la clase más común en servicios de red. Tu traza dice que el span tardó 800 ms. Abres el flame graph de CPU de esos 800 ms y encuentras 12 ms repartidos en ruido. No hay contradicción ni bug: el proceso no estaba ejecutando nada. Estaba esperando -una consulta a la base de datos, un lock, una escritura a disco, el turno del planificador porque el cgroup agotó su cuota de CPU- y un profiler que muestrea la CPU no puede muestrear a nadie que no esté en la CPU.

El perfil que responde a esa pregunta se llama off-CPU y se construye al revés: en lugar de interrumpir periódicamente, se engancha al punto del planificador del kernel donde un hilo abandona la CPU, guarda la pila y el instante, y cuando ese mismo hilo vuelve a ejecutarse calcula cuánto estuvo fuera. El resultado no es «cuántas muestras cayeron aquí» sino «cuántos microsegundos se bloqueó aquí», y eso mide directamente esperas de E/S, contención de locks y todo lo que el planificador considera sueño.

```
# bcc trae la herramienta hecha. Instalacion:
# Debian/Ubuntu: apt-get install bpfcc-tools
# RHEL/CentOS: yum install bcc-tools

# 30 segundos de tiempo off-CPU de un PID, en pilas plegadas (folded stacks),
# con pila de usuario y de kernel. El umbral -m descarta bloqueos triviales.
sudo /usr/share/bcc/tools/offcputime -df -p $(pgrep -f mi-servicio) -m 1000 30 \
> /tmp/off.folded

# Convertir a flame graph (FlameGraph de Brendan Gregg)
./flamegraph.pl --colors=io --title="Off-CPU 30s" /tmp/off.folded > /tmp/off.svg
```

Tres cosas que conviene saber antes de leer el resultado. La primera: el eje X de un flame graph off-CPU son microsegundos bloqueados, no muestras, así que un hilo dormido que no molesta a nadie -un pool de conexiones esperando trabajo, un temporizador- dominará el gráfico y no significa nada. Por eso casi siempre quieres filtrar por hilo o por rango de duración en lugar de mirar el agregado. La segunda: offcputime cobra por evento, no por intervalo, y un servicio con cientos de miles de cambios de contexto por segundo puede notarlo; el umbral -m y un -p concreto son lo que lo mantiene barato. La tercera: si el proceso está bloqueado en una llamada al sistema y no tienes punteros de marco en la biblioteca C, la pila se corta justo donde empieza a ser interesante -lo que nos lleva a la sección siguiente.

El perfil de reloj de pared: sumar los dos en un solo gráfico

Lo que de verdad quieres ver, casi siempre, es la suma: wall clock = on-CPU + off-CPU. Si capturas ambos en la misma ventana y normalizas las escalas -las muestras de CPU a 99 Hz representan aproximadamente 10,1 ms cada una; el off-CPU ya viene en microsegundos- puedes fusionar las pilas plegadas y obtener un único flame graph donde la anchura de una rama es tiempo real transcurrido. Es la vista que convierte «el servicio está lento» en «el 68% del tiempo de esta petición se va esperando a

Postgres desde este método».

```
#!/usr/bin/env python3
"""Fusiona pilas plegadas on-CPU y off-CPU en una escala comun de microsegundos.

on.folded : salida de 'profile -f' (muestras a HZ hercios)
off.folded : salida de 'offcputime -df' (microsegundos bloqueados)
"""
import sys
from collections import defaultdict

HZ = 99 # frecuencia de muestreo usada para el perfil on-CPU
US_POR_MUESTRA = 1_000_000 / HZ

def cargar(ruta, factor):
total = defaultdict(float)
with open(ruta) as fh:
for linea in fh:
linea = linea.strip()
if not linea:
continue
pila, _, valor = linea.rpartition(" ")
if not pila:
continue
total[pila] += float(valor) * factor
return total

def main(on_path, off_path):
combinado = defaultdict(float)
# Marcamos cada rama para poder distinguirlas por color en el flame graph
for pila, us in cargar(on_path, US_POR_MUESTRA).items():
combinado[pila + ";[on-cpu]"] += us
for pila, us in cargar(off_path, 1.0).items():
combinado[pila + ";[off-cpu]"] += us

total = sum(combinado.values()) or 1.0
for pila, us in sorted(combinado.items(), key=lambda kv: -kv[1]):
# flamegraph.pl espera enteros; redondeamos a microsegundos
sys.stdout.write(f"{pila} {int(us)}\n")
print(f"# total = {total/1e6:.2f} s de reloj de pared", file=sys.stderr)

if __name__ == "__main__":
main(sys.argv[1], sys.argv[2])
```

Con eso montado, el diagnóstico cambia de naturaleza. Una meseta ancha con el sufijo [off-cpu] bajo una llamada a psycopg no es «la base de datos va lenta»: es una cifra concreta de segundos de reloj que puedes comparar con el pg_stat_statements del otro lado. Y una meseta ancha [off-cpu] bajo futex_wait con pilas de tu propio código a ambos lados es contención de locks, que es un problema tuyo y no de nadie más.

En lenguajes concretos tienes atajos que evitan montar todo esto. En Go, los perfiles block y mutex que ya vimos cubren la parte de sincronización del off-CPU sin salir del proceso. En Java, asprof -e wall -t te da directamente el perfil de reloj de pared por hilo. En Python, py-spy record --idle incluye los hilos que están esperando, que por defecto se descartan -y ese único parámetro es la diferencia entre un flame graph que explica la latencia y uno que insiste en que tu servicio no hace nada.Símbolos: por qué tu flame graph está lleno de direcciones hexadecimales

Símbolos: por qué tu flame graph está lleno de direcciones hexadecimales

El primer flame graph que saca casi todo el mundo con un perfilador de sistema es decepcionante: una torre de barras que dicen 0x7f3a91c4e210 y poco más. No es un fallo de configuración del profiler; es que el perfilado nativo tiene dos problemas distintos que se resuelven por separado, y confundirlos hace perder días. El primero es desenrollar la pila: averiguar la cadena de direcciones de retorno. El segundo es simbolizar: traducir cada dirección a un nombre de función, un fichero y una línea.

Desenrollar: punteros de marco frente a DWARF

La forma barata de desenrollar es el puntero de marco. Si el compilador reserva el registro rbp para encadenar los marcos, recorrer la pila es seguir una lista enlazada, y eso es exactamente lo que hace el ayudante del kernel bpf_get_stackid: unas decenas de nanosegundos, dentro del contexto de la señal, sin tocar memoria de usuario arbitraria. El problema es que durante quince años la práctica habitual fue compilar con -fomit-frame-pointer para recuperar un registro, y ese registro valía en torno a un 1% de rendimiento. A cambio, todo el ecosistema perdió la capacidad de perfilarse.

La alternativa es DWARF. Los binarios ELF llevan una sección .eh_frame -creada originalmente para el desenrollado de excepciones de C++ y después reutilizada como formato general- que describe, para cada dirección del programa, cómo reconstruir el marco del llamante. Es información completa y es información cara: .eh_frame contiene un intérprete de bytecode, algo que no puedes ejecutar dentro de un programa eBPF con su límite de instrucciones y sin bucles no acotados. La solución que adoptaron los profilers modernos es precalcular: se lee .eh_frame en espacio de usuario, se aplana a una tabla de stack deltas -para cada rango de direcciones, qué registro es la base y qué desplazamiento tiene la dirección de retorno- y se carga esa tabla en mapas eBPF para que el desenrollador del kernel sólo tenga que hacer búsquedas binarias.

La buena noticia es que la industria dio marcha atrás. Fedora 38 y Ubuntu 24.04 reactivaron los punteros de marco en sus paquetes por defecto, y a partir de ahí el camino rápido vuelve a funcionar para la mayoría del software de distribución. La regla práctica para tus propios binarios es simple: en Go ya los tienes; en Rust y C++ actívalos explícitamente y acepta el 1%.

```
# Rust: punteros de marco en el perfil de release, sin tocar Cargo.toml
RUSTFLAGS="-C force-frame-pointers=yes" cargo build --release

# C/C++
cc -O2 -g -fno-omit-frame-pointer -fasynchronous-unwind-tables ...

# Comprobar que un binario trae lo necesario para desenrollar
readelf -S ./mi-binario | grep -E 'eh_frame|debug_'
# Y si el prologo empieza por 'push %rbp; mov %rsp,%rbp', hay punteros de marco:
objdump -d ./mi-binario | grep -A2 '<main>:' | head -3
```

Simbolizar: dónde viven los nombres y por qué no están donde perfilas

Desenrollada la pila, hacen falta los nombres. Y aquí entra la práctica universal de las imágenes de contenedor: strip. Una imagen de producción bien hecha no lleva tabla de símbolos, porque pesa y porque no hace falta para ejecutar. El profiler, que corre como DaemonSet en el nodo, encuentra el binario pero no encuentra los nombres.

Hay tres salidas, y elegir la correcta depende de tu pipeline de build. La primera es simbolizar en el origen: el agente sube las direcciones sin resolver al backend y el backend las resuelve contra un almacén de símbolos al que subiste los ficheros de depuración en el momento del build. Es lo que hacen Polar Signals y Parca, y es la opción sana si controlas el CI. La segunda es debuginfod: un

protocolo HTTP sencillo que sirve ficheros de depuración indexados por Build ID, el identificador único que el enlazador deja en la sección .note.gnu.build-id de cada binario. Fedora, Debian y Ubuntu operan servidores públicos para sus paquetes, y montar uno propio para tus artefactos son dos horas de trabajo. La tercera, la peor, es dejar los símbolos dentro de la imagen y pagar el tamaño.

```
# En el build: separar los simbolos en un fichero aparte y dejar un enlace
# por Build ID dentro del binario. La imagen queda pequena y los simbolos
# siguen siendo recuperables.
objcopy --only-keep-debug mi-binario mi-binario.debug
objcopy --strip-debug mi-binario
objcopy --add-gnu-debuglink=mi-binario.debug mi-binario

# El Build ID es la clave con la que se recuperan despues
readelf -n mi-binario | grep -A1 'Build ID'
# -> Build ID: 4a7f1c9e2b5d8a3f6c0e9b1d4a7f1c9e2b5d8a3f

# Publicar en un almacen indexado por Build ID (layout de debuginfod)
BID=$(readelf -n mi-binario | awk '/Build ID/{print $3}')
install -D mi-binario.debug "/srv/debuginfo/${BID:0:2}/${BID:2}.debug"

# En la maquina donde simbolizas
export DEBUGINFOD_URLS="https://debuginfod.interno/ https://debuginfod.elfutils.org/"
debuginfod-find debuginfo "$BID" # descarga y cachea en ~/.cache/debuginfod_client
```

Los lenguajes con runtime tienen su propio conjunto de trampas. Go embebe su tabla de símbolos en el binario, así que casi siempre funciona sin hacer nada -salvo que alguien haya puesto -ldflags="-s -w" en el Dockerfile para ahorrar seis megas, que es la causa número uno de perfiles de Go ilegibles. Los lenguajes interpretados son un problema distinto y más profundo: la pila nativa de un proceso Python muestra _PyEval_EvalFrameDefault repetido veinte veces, porque las funciones Python no son marcos nativos. Por eso los perfiladores de sistema necesitan unwinders específicos por runtime, que leen las estructuras internas del intérprete para reconstruir la pila del lenguaje. El perfilador eBPF de OpenTelemetry, donado por Elastic y ya componente oficial del Collector, incorpora soporte de runtime para Go, Node.js (V8), Ruby, .NET 9 y 10, y un primer soporte para BEAM (Erlang y Elixir). Cada uno de esos unwinders depende de compensaciones internas de la versión del runtime, lo que explica la regla que parece arbitraria y no lo es: si actualizas el intérprete, actualiza el perfilador, o los símbolos se convertirán en basura silenciosamente.Perfilar en CI: convertir una regresión de CPU en un fallo de build

Perfilar en CI: convertir una regresión de CPU en un fallo de build

El continuous profiling te dice que una regresión llegó a producción. Perfilar en CI te dice que no llegue. Son cosas distintas y las dos valen la pena, pero la segunda se monta mal casi siempre, porque la gente intenta poner un umbral absoluto («este benchmark debe tardar menos de 5 ms») sobre unos runners compartidos cuyo rendimiento varía un 30% según el vecino. El resultado es un test que falla los martes y que acaba marcado como continue-on-error, que es una forma elegante de borrarlo.

La forma que funciona tiene tres reglas. Una: comparar, no medir. Ejecuta la rama base y la rama del PR en el mismo trabajo, en la misma máquina, intercalados; lo que te importa es la diferencia relativa, y esa sí es estable. Dos: varias repeticiones y estadística, no una cifra. benchstat aplica un test de Mann-Whitney y sólo reporta un cambio cuando es significativo; una única ejecución no distingue una regresión del 8% del ruido. Tres: guarda el perfil, no sólo el número. Que el build falle diciendo «+14% de CPU» es útil; que además adjunte el pprof diferencial que señala la función responsable es lo que hace que alguien lo arregle esa misma tarde en lugar de abrir un ticket.

```
#!/usr/bin/env bash
# ci/perf-gate.sh - falla el build si un benchmark empeora de forma significativa
set -euo pipefail

BASE="${1:-origin/main}"
PKG="${2:-./internal/pipeline}"
N=10 # repeticiones por lado
UMBRAL_PCT=5 # tolerancia

trabajo=$(mktemp -d); trap 'rm -rf "$trabajo"' EXIT

medir () { # $1 = etiqueta
go test "$PKG" -run '^$' -bench . -benchmem \
-count "$N" -cpuprofile "$trabajo/$1.pprof" \
| tee "$trabajo/$1.txt"
}

git stash push --include-untracked --quiet || true
git checkout --quiet "$BASE"
medir base
git checkout --quiet -
git stash pop --quiet || true
medir pr

# Comparacion estadistica: benchstat solo reporta diferencias significativas
benchstat "$trabajo/base.txt" "$trabajo/pr.txt" | tee "$trabajo/benchstat.txt"

# Perfil diferencial: que funcion se comio el tiempo extra
go tool pprof -top -nodecount=15 \
-base "$trabajo/base.pprof" "$trabajo/pr.pprof" \
| tee "$trabajo/diff.txt"

peor=$(awk '/sec\/op/ {if (match($0, /\+([0-9.]+)%/, m) && m[1]+0 > max) max=m[1]+0} END {print
max+0}' \
"$trabajo/benchstat.txt")

if (( $(echo "$peor > $UMBRAL_PCT" | bc -l) )); then
echo "::error::Regresion de rendimiento: +${peor}% (umbral ${UMBRAL_PCT}%)"
cp "$trabajo"/{benchstat.txt,diff.txt,pr.pprof,base.pprof} "${ARTIFACTS_DIR:-.}/"
exit 1
fi
echo "Sin regresion significativa (peor caso: +${peor}%)"
```

La opción -base de pprof es la pieza que casi nadie usa y la que más tiempo ahorra: resta un perfil de otro y te deja únicamente el delta, así que en lugar de comparar dos flame graphs a ojo -tarea en la que el ojo humano es sorprendentemente malo- ves directamente las cinco funciones que aparecieron o crecieron. Existe también -diff_base, que normaliza por el total antes de restar; usa -base cuando quieras ver crecimiento absoluto y -diff_base cuando la carga total haya cambiado entre las dos ejecuciones.

Una advertencia para no crear una religión con esto: un test de rendimiento en CI cubre el camino caliente que tú decidiste medir, con los datos que tú generaste. Una regresión que sólo aparece con la distribución real de tamaños de payload de producción no la vas a ver aquí. El perfilado continuo y el perfilado en CI no compiten; el segundo evita las regresiones obvias y el primero encuentra las que no lo son.

Lo que cuesta, y cómo no pagarlo dos veces

El continuous profiling tiene dos costes y casi todo el mundo presupuesta sólo uno. El coste en el proceso perfilado ya lo hemos visto y es pequeño: entre el 2% y el 5% de CPU con un agente eBPF,

menos del 1% con JFR en configuración por defecto, unas decenas de megabytes de memoria por pod. El segundo coste es el del backend, y ese sí escala con decisiones que tomas sin darte cuenta.

La forma del precio en los servicios gestionados lo explica bien. Grafana Cloud Profiles, por ejemplo, factura tres conceptos distintos -procesado por gigabyte, escritura por gigabyte y retención por gigabyte y mes-, con el plan gratuito en torno a 50 GB al mes y 14 días de retención, y el de pago con 30 días. Los despliegues reales superan la estimación inicial con una regularidad monótona, y la causa casi siempre es la misma que en Prometheus: la cardinalidad. Cada combinación distinta de etiquetas es una serie de perfiles independiente que se almacena, se indexa y se retiene por separado. Añadir version con el SHA del despliegue es una de las mejores decisiones que puedes tomar, porque convierte una regresión en algo atribuible; añadir request_id es el mismo error que ya cometiste una vez con las métricas, sólo que los perfiles pesan tres órdenes de magnitud más que un contador.

Las cuatro palancas que de verdad mueven la factura, en orden de efecto: qué servicios perfilas -la práctica habitual en organizaciones grandes no es perfilarlo todo, sino los servicios caros y los que están en la ruta crítica; un CronJob que corre tres minutos al día no necesita perfilado continuo-; la frecuencia de muestreo, donde bajar de 99 Hz a 19 Hz reduce el volumen casi cinco veces y sigue siendo perfectamente utilizable para tendencias a lo largo de días, aunque ya no para diagnosticar una función concreta; la retención por niveles, guardando alta resolución dos semanas y agregados a baja resolución un año, que es lo que necesitas para responder «¿cuándo empezó a subir esto?»; y el número de tipos de perfil activos, porque activar CPU, heap, allocations, goroutines, mutex y block a la vez multiplica el ingreso por seis cuando en la práctica CPU y heap responden al 80% de las preguntas.

Del lado autogestionado, Pyroscope 2.0 -publicado el 21 de abril de 2026- rehízo precisamente la parte de almacenamiento y consulta que se había vuelto cara y operativamente incómoda en el diseño original, así que si evaluaste Pyroscope hace un año y lo descartaste por coste de operación, la evaluación está obsoleta. Y si tu criterio es no depender de un proveedor, el punto de entrada razonable en 2026 es el formato: OTLP profiles hace round-trip sin pérdidas contra pprof, de modo que los datos que recojas hoy con un SDK siguen siendo tuyos y convertibles mañana.Un caso completo: el 38% de CPU que nadie había pedido

Un caso completo: el 38% de CPU que nadie había pedido

Conviene ver cómo encajan las piezas en un problema real, porque en un incidente nunca se usa una sola herramienta. Lo que sigue es un caso compuesto pero representativo: un servicio de ingesta en Go, doce réplicas, alrededor de 4.000 peticiones por segundo, que después de una actualización empezó a necesitar dieciséis réplicas para el mismo tráfico. Nada estaba caído. La latencia p99 había pasado de 82 ms a 104 ms, por debajo del SLO. La factura de cómputo había subido un 33%.

Lo que dijeron las métricas. CPU por réplica al 71% frente al 48% anterior. Peticiones por segundo idénticas. Sin cambios en el tamaño de los payloads. El GC de Go corriendo más a menudo, pero con pausas normales. Las métricas acotaron el problema -es CPU, no es E/S, no es una dependencia- y ahí se agotaron, que es exactamente lo que se espera de ellas.

Lo que dijeron las trazas. El span handle_batch había pasado de 31 ms a 44 ms de media. Dentro de él no había hijos instrumentados. Trece milisegundos aparecidos dentro de un span sin estructura interna: la traza había hecho su trabajo y no podía hacer más.

Lo que dijo el perfil. Con Pyroscope ya recogiendo perfiles etiquetados por version, la consulta fue

directa: perfil de CPU del despliegue actual menos perfil de CPU del despliegue anterior, misma ventana horaria del mismo día de la semana. El diferencial señaló una función que nadie había tocado, encoding/json.Marshal, con un 38% del tiempo de CPU frente al 11% anterior. Subiendo por la pila, el llamante nuevo era una función de validación añadida en la actualización que, para comprobar que un campo opcional cumplía un esquema, serializaba la estructura completa a JSON -incluido un campo de metadatos que podía tener decenas de kilobytes- y luego la descartaba.

```
# Lo que se ejecuto, reducido a lo esencial. Los dos perfiles se descargan
# del backend por etiqueta de version; -base deja solo el delta.
pprof-cli download --service ingesta --profile cpu \
--from '2026-09-08T14:00Z' --to '2026-09-08T14:30Z' \
--selector 'version="v2.14.0"' -o previo.pprof
pprof-cli download --service ingesta --profile cpu \
--from '2026-09-15T14:00Z' --to '2026-09-15T14:30Z' \
--selector 'version="v2.15.0"' -o actual.pprof

go tool pprof -top -nodecount=12 -base previo.pprof actual.pprof
# flat flat% sum% cum cum%
# 4.21s 38.1% 38.1% 5.02s 45.4% encoding/json.(*encodeState).marshal
# 0.63s 5.7% 43.8% 5.65s 51.1% ingesta/validate.CheckOptionalSchema
# 0.29s 2.6% 46.4% 0.29s 2.6% runtime.mallocgc
# -0.11s -1.0% 45.4% -0.11s -1.0% ingesta/decode.Fast
```

El arreglo fue de cuatro líneas: comprobar la presencia del campo antes de validarlo y validar sólo el subárbol relevante en lugar de la estructura entera. Pero la parte interesante del caso no es el arreglo, es la cronología. Sin perfiles, esto se habría resuelto -se resolvió así en el mismo equipo un año antes con un problema análogo- añadiendo réplicas, abriendo un ticket de «investigar consumo de CPU» y encontrándolo tres semanas después mientras se miraba otra cosa. Con perfiles y la etiqueta de versión, desde la alerta de coste hasta la línea culpable pasaron once minutos, y nueve de ellos fueron esperar a que alguien con permisos abriera el panel.

Hay dos lecciones operativas ahí, y ninguna es sobre JSON. La primera: la etiqueta con el SHA o la versión del despliegue es lo que convierte un perfil en un instrumento de atribución; sin ella habrías visto que json.Marshal consume mucho -lo cual es cierto en casi cualquier servicio de Go y no te dice nada- en lugar de ver que creció. La segunda: el problema nunca disparó una alerta de latencia, porque no rompió el SLO. Lo detectó el presupuesto de cómputo. Un porcentaje razonable de las regresiones que encuentra el perfilado continuo son de esta clase: no rompen nada, sólo cuestan dinero para siempre, y son invisibles para un sistema de observabilidad que sólo vigila umbrales de error y latencia.

Ocho errores recurrentes

1. Exponer pprof en el mux público. Ya cubierto, pero merece repetirse porque sigue siendo el fallo más frecuente: import _ "net/http/pprof" más http.ListenAndServe(":8080", nil) es un volcado de memoria y una negación de servicio accesibles desde internet.
2. Perfilar cinco segundos y sacar conclusiones. Con pocas muestras, un 5% puede ser un 3% o un 7%. Si tu "optimización" mueve la aguja menos de un 2%, no has medido nada.
3. Leer el eje X de un flame graph como si fuera tiempo. No lo es. Para secuencia temporal necesitas una traza de ejecución.
4. Confundir alloc_space con inuse_space. Optimizar la función que más ha asignado desde el arranque cuando lo que tienes es una fuga es tiempo perdido con una sensación de progreso.
5. Dejar mutex y block desactivados. Son los valores por defecto de Go, y significan que las

regresiones por contención -una fracción enorme del total- son invisibles para ti.

6. Perfilar con eBPF sin frame pointers. Obtienes pilas truncadas y pierdes una semana culpando a la herramienta. Verifica los flags de compilación de tu imagen base antes de desplegar el agente.
7. Meter identificadores en las etiquetas. user_id, request_id o la URL con IDs dentro hacen con el backend de perfiles exactamente lo mismo que hacen con Prometheus: lo funden.
8. Comparar dos ventanas con carga distinta. Un diferencial entre las 04:00 y las 12:00 muestra que todo creció. No es una regresión: es tráfico. Normaliza por unidad de trabajo o compara ventanas equivalentes.

Checklist de producción

- Los endpoints de pprof (o equivalentes) escuchan en loopback o en un puerto de administración que no está en el Service ni en el Ingress.
- El WriteTimeout del listener de administración es mayor que la duración máxima de perfil que vas a pedir.
- En Go, SetMutexProfileFraction y SetBlockProfileRate están activados con valores conservadores y medidos en un entorno de carga antes de ir a producción.
- Las imágenes base están compiladas con -fno-omit-frame-pointer si vas a usar un agente de eBPF.
- Toda muestra lleva, como mínimo, las etiquetas service, env, region y version (SHA del commit).
- Ninguna etiqueta contiene identificadores de usuario, de petición ni URLs sin normalizar.
- Existe una política de retención explícita: alta resolución días, agregados semanas o meses.
- La sobrecarga del agente está medida en tu carga real, no asumida desde el README, y hay una alerta si el agente deja de reportar.
- El runbook de incidentes incluye el paso "abre el flame graph diferencial contra el despliegue anterior" antes de "haz rollback a ciegas".
- Los perfiles se tratan como datos potencialmente sensibles: los nombres de función y de fichero revelan estructura interna, y en algunos casos los argumentos. Control de acceso acorde.

Preguntas frecuentes

¿Cuánto cuesta de verdad tener esto encendido siempre? Con un agente de muestreo moderno, del orden de un 2-5% de CPU y menos de 50 MB de memoria por pod. El coste dominante no es el agente sino el almacenamiento, y ese lo controlas con la retención y con la disciplina de cardinalidad.

¿Puedo perfilar sólo cuando haya un problema? Puedes, y es mejor que nada, pero pierdes lo más valioso: la línea base. Sin histórico no tienes con qué comparar, y un flame graph aislado te dice en qué gasta CPU tu servicio, no qué cambió.

¿Perfil o traza de ejecución? El perfil responde "dónde se gastó la CPU". La traza de ejecución responde "por qué no se estaba gastando": esperas, bloqueos, pausas de GC, planificación. Si tu p99 es alto pero la CPU está ociosa, tu respuesta está en la traza, no en el perfil.

¿Merece la pena en un servicio dominado por E/S? Sí, pero cambiando el modo. Un perfil de CPU en un servicio que pasa el 95% del tiempo esperando a Postgres no muestra gran cosa. Perfila en modo

reloj de pared (oncpu=False en Pyroscope, --mode=wall en Tachyon, el modo por defecto de py-spy) y verás dónde está la espera.

¿Y si mi lenguaje no está en la lista? Casi cualquier runtime con soporte de perf puede alimentar un flame graph mediante el formato collapsed stacks, que es una línea de texto por pila con su contador. Es el mínimo común denominador de todo el ecosistema.

¿Espero a que OpenTelemetry Profiles llegue a GA? No. Monta continuous profiling ahora con lo que funciona y ve evaluando la señal OTel en paralelo. Cuando madure, migrar será cambiar de exportador; lo que no se recupera es el año de histórico que no recogiste.

Glosario

- Profiler de muestreo - interrumpe o inspecciona el programa a intervalos fijos y registra la pila. Coste proporcional a la frecuencia, no al trabajo.
- Profiler determinista (o de trazado) - instrumenta cada llamada. Exacto en conteos, caro y sesgado hacia código con muchas llamadas pequeñas.
- Flame graph - visualización agregada de pilas: anchura igual a muestras, altura igual a profundidad de pila, eje X sin significado temporal.
- Self time (flat) - tiempo en una función excluyendo sus llamadas. Es lo que aparece como meseta en la parte alta del flame graph.
- Cumulative time (cum) - tiempo en una función incluyendo todo lo que llama.
- pprof - formato de perfiles de Google (protobuf) y la herramienta que lo lee. El estándar de facto del sector.
- Collapsed stacks - formato de texto con una pila por línea, separada por punto y coma, seguida del número de muestras. El intercambio universal entre herramientas.
- Frame pointer - registro que apunta al marco de pila anterior. Sin él, desenrollar la pila desde el kernel es inviable y el perfil sale truncado.
- eBPF - máquina virtual del kernel de Linux que permite ejecutar programas verificados en puntos de enganche del kernel. La base de los profilers sin agente en el proceso.
- Span profile - muestras de perfil etiquetadas con el identificador del span activo, lo que permite saltar de una traza lenta a sus pilas.
- Flight recorder - tracer que mantiene en memoria una ventana circular reciente y la vuelca sólo cuando el programa detecta una anomalía.

Por dónde empezar

Si no tienes nada montado, el orden que da resultados más rápido es este. Primero, expón pprof (o el equivalente de tu lenguaje) en un puerto de administración en un solo servicio, el que más CPU consuma de tu factura. Segundo, captura una línea base de 60 segundos y guárdala en el repositorio junto a la versión. Tercero, despliega un agente continuo en ese mismo servicio con cuatro etiquetas: service, env, region y version. Cuarto, la próxima vez que despliegues, abre el diferencial contra el SHA anterior.

Ese cuarto paso es el que convence al resto del equipo, porque es la primera vez que alguien puede señalar una función concreta y decir "esto costaba 12 ms por petición ayer y hoy cuesta 40". Todo lo

demás -la cobertura de la flota entera, el enlace con trazas, la migración a OpenTelemetry- viene después y es mucho más fácil de justificar.

ENGLISH VERSION

Continuous Profiling in Production: Flame Graphs, pprof, py-spy, eBPF, and OpenTelemetry's Fourth Signal

The gap metrics, logs, and traces leave behind

You have a dashboard telling you the p99 on /checkout went from 180 ms to 640 ms on Tuesday afternoon. You have traces pointing at the render_cart span as the culprit for nearly all of it. You have logs correlated by request_id. And you still don't know what to change, because none of those three signals tells you which line of code is eating those milliseconds.

That is precisely the gap profiling fills. A metric tells you how much. A trace tells you where, at whatever resolution someone bothered to instrument. A profile tells you what: which function, which line, which complete call stack was burning CPU or allocating memory while the clock ran. It is the only signal that doesn't depend on you having anticipated the problem when you instrumented.

For years profiling was a laboratory activity: reproduce the problem on your laptop, run a profiler, stare at a graph. That works for an algorithm and fails for almost everything else. Production load doesn't reproduce, production data doesn't reproduce, and production hardware doesn't either. Continuous profiling - profiling the whole fleet, all the time, at an overhead you can afford - turns that laboratory activity into just another observability signal, with history, with version-to-version comparison, and with the ability to answer questions after the incident is over.

This guide is about setting that up without breaking anything: the minimum statistics you need to avoid drawing false conclusions, how to actually read a flame graph, correct configuration in Go, Python, and Node.js, the jump to continuous profiling with Pyroscope and eBPF, and the real state of the OpenTelemetry profiles signal today.

Sampling versus instrumentation: why a production profiler has to approximate

There are two ways to build a profiler. A deterministic (tracing) profiler intercepts every function entry and exit and times them. You get exact call counts and overhead ranging from 30% to well over 1000%, because the cost of instrumentation is proportional to the number of calls, not to runtime. Python's cProfile and V8's --prof belong to this family.

A sampling profiler intercepts nothing. Every N milliseconds it interrupts (or reads from outside) the process, captures the call stack, and records it. The cost is proportional to sampling frequency and thread count, not to the work your program does, which is why it is constant and predictable. Every profiler you can leave running in production is a sampling profiler.

The price is that the result is a statistical estimate. It's worth doing the arithmetic once, because it decides how long you have to profile before you should believe the answer.

At 10 kHz for 10 seconds you collect roughly 100,000 samples. If a function appears in 5,000 of them, you estimate it consumed 5% of the time, about 500 ms. The margin of error on that figure with 100,000

samples is roughly ±0.5%. With only 1,000 samples - say, 100 Hz for 10 seconds - that same 5% could actually be anywhere from 3% to 7%. The rule of thumb: a difference under 2% is not signal, it's noise, unless you have accumulated an enormous number of samples.

Two consequences follow, and people usually learn them the hard way:

- Don't compare 5-second profiles. For a comparison between two versions, accumulate minutes, not seconds. With continuous profiling this is free, because the backend aggregates samples across the fleet and across the whole time window.
- Very short functions disappear. A function that takes 20 µs and runs a thousand times per second will show up (20 ms/s of accumulated CPU). One that takes 20 µs and runs three times a day never will. Sampling measures aggregate time, not events.

One small detail that does matter: pick prime or odd frequencies such as 99 Hz or 97 Hz rather than 100 Hz. Plenty of systems have periodic work anchored to multiples of 10 ms - kernel timers, application tickers, GC pacing. Sample at exactly 100 Hz and you risk locking in phase with that work and either over-representing it or never seeing it at all. At 99 Hz the phase drifts and the bias vanishes. That's why the canonical perf record example uses -F 99.

How to read a flame graph without fooling yourself

The flame graph is the standard visualization for a profile, and also the most misread. Three rules are enough to read one properly:

1. The X axis is not time. Nearly everyone assumes it is, and it isn't. Boxes are sorted alphabetically within each level so identical stacks merge. A flame graph has no timeline: if you want the sequence of events, you need an execution trace, not a profile.
2. Width is sample count, meaning accumulated time. A box occupying 30% of the width means that stack was active in 30% of the samples. No more, no less.
3. Flat plateaus are the target. A wide box with wide children only means it calls something expensive; the real work is further down. A wide box with no children - a plateau near the top of the graph - is self time: that's where CPU is actually being burned. When someone says "look for the plateaus," this is what they mean.

The differential flame graph is the tool that makes profiling operational. Take a profile from the previous version as a baseline and one from the new version, and the graph colors growth in red and shrinkage in blue. This is how you localize a performance regression in minutes instead of days. The trap: if the two windows carry different load, everything grows or everything shrinks and the diff means nothing. Always normalize by a comparable unit of work (CPU per request, not total CPU) or compare windows with equivalent traffic.

Finally, an icicle graph is the same graph drawn downward from the root. Pyroscope and pprof's web viewer use this orientation by default. Semantically nothing changes; the root is just on top.

Go: pprof, wired up correctly

Go ships the best built-in profiler of any mainstream language, and also the easiest way to expose your process to the internet without noticing. Let's start there.

The trick everyone copies is import _ "net/http/pprof". That blank import has a side effect: it registers the pprof handlers on http.DefaultServeMux. If your public server also uses the DefaultServeMux - and it does if you call http.ListenAndServe(":8080", nil) - you have just published /debug/pprof/heap on the internet. That is a memory dump complete with function names, and /debug/pprof/profile?seconds=3600 is a free denial of service on top.

The correct approach is to import the package by name and register the handlers on your own admin mux, listening on loopback or on a port you never publish:

```
package main

import (
"log"
"net/http"
"net/http/pprof"
"runtime"
"time"
)

func main() {
// Mutex contention: record 1 in every 5 events. 0 disables it.
runtime.SetMutexProfileFraction(5)

// Blocking: the runtime aims for one sample per 10,000 ns
// (10 µs) of time spent blocked. 0 disables it.
runtime.SetBlockProfileRate(10_000)

// Heap: one sample per 512 KiB allocated. This is the default;
// lowering it gives more detail and costs more CPU.
runtime.MemProfileRate = 512 * 1024

// pprof lives on its own mux, never on the public one.
admin := http.NewServeMux()
admin.HandleFunc("/debug/pprof/", pprof.Index)
admin.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
admin.HandleFunc("/debug/pprof/profile", pprof.Profile)
admin.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
admin.HandleFunc("/debug/pprof/trace", pprof.Trace)

go func() {
srv := &http.Server{
Addr: "127.0.0.1:6060",
Handler: admin,
ReadTimeout: 5 * time.Second,
// A 60 s CPU profile must not die on a write timeout.
WriteTimeout: 150 * time.Second,
}
log.Fatal(srv.ListenAndServe())
}()

// ... your public server here, on a different mux and port.
}
```

That WriteTimeout is silent failure number one: you ask for ?seconds=60, the server cuts the connection at 30, and you sit there staring at a truncated profile with nothing explaining why.

The six profiles and the question each one answers

- profile - CPU. Sampled at 100 Hz per running goroutine. Answers: where is CPU time going? This is the one you want 80% of the time.
- heap - memory. Answers two different questions depending on which index you look at: inuse_space is what is still live right now (leaks), alloc_space is everything allocated since startup

(GC pressure). Mixing them up is the classic mistake covered below.

- goroutine - how many goroutines exist and where they are parked. The first thing to check on a suspected goroutine leak.
- mutex - where time is lost waiting on a sync.Mutex. Off by default.
- block - where goroutines block: channels, select, WaitGroup, syscalls. Off by default.
- allocs - an alias for heap with the default index set to alloc_space.

The operational advice: leave mutex and block profiling permanently enabled in your continuous profiler, with conservative fractions like the ones above. A large share of performance regressions aren't CPU at all, they're contention - and those are exactly what the CPU profile can't show you, because a goroutine waiting on a lock consumes no CPU and therefore appears in no sample.

The tooling workflow

```
# A 30-second CPU profile, opened in the interactive web viewer.
go tool pprof -http=:8080 http://127.0.0.1:6060/debug/pprof/profile?seconds=30

# Memory live right now (leaks).
go tool pprof -http=:8080 -sample_index=inuse_space \
http://127.0.0.1:6060/debug/pprof/heap

# Total volume allocated (GC pressure).
go tool pprof -http=:8080 -sample_index=alloc_space \
http://127.0.0.1:6060/debug/pprof/heap

# Save a baseline and diff against it after a change.
curl -o base.pprof "http://127.0.0.1:6060/debug/pprof/profile?seconds=60"
# ... deploy the change ...
curl -o new.pprof "http://127.0.0.1:6060/debug/pprof/profile?seconds=60"
go tool pprof -http=:8080 -base base.pprof new.pprof
```

Inside the viewer, the views people actually use are Flame Graph to orient yourself, Top sorted by flat (self time) for the suspect list, and Source for a line-by-line breakdown of one specific function. The -base mode is the differential flame graph: red for growth, blue for shrinkage.

Labels: attributing CPU per tenant, per route, or per version

An aggregate profile of a multi-tenant service tells you 40% of CPU goes into JSON deserialization. Half useful. What you want to know is whether that 40% is spread across a thousand customers or whether one of them is sending 30 MB requests. pprof labels solve that:

```
import (
"context"
"net/http"
"runtime/pprof"
)

func profileLabels(next http.Handler) http.Handler {
return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
labels := pprof.Labels(
"tenant", tenantFromRequest(r),
"route", routePattern(r), // the pattern, not the URL with IDs
)
pprof.Do(r.Context(), labels, func(ctx context.Context) {
next.ServeHTTP(w, r.WithContext(ctx))
})
})
```

```
}
```

Two details that cost an afternoon if you don't know them. First: only the CPU profile records labels; the heap profile does not carry them, so you cannot attribute memory per tenant this way. Second: labels are inherited by goroutines you launch after applying them within the same context, but they are not applied retroactively to a worker pool that was already running. If you fan out to a pre-existing pool, the heavy work will show up unlabeled.

And one hygiene rule: use the route pattern (/users/{id}), not the concrete URL. Exactly as in Prometheus, a label containing IDs will destroy the cardinality of your profiling backend.

alloc_space versus inuse_space, the mistake that sends you to the wrong function

A service is using 4 GB and you think you have a leak. You open the heap profile, see json.Unmarshal at 300 GB, and start optimizing deserialization. But you were looking at alloc_space, which is the cumulative total since process start: 300 GB spread over six days of uptime is 2 GB per hour, entirely normal and all of it reclaimed by the GC. The leak is somewhere else, and only inuse_space can see it.

The short version: inuse_space answers "who is retaining memory?" and is for leaks; alloc_space answers "who is generating garbage?" and is for GC pressure and latency. Both questions are legitimate; answering them with the wrong index is not.

When the problem isn't CPU: Go 1.25's FlightRecorder

There is a class of problem no CPU profile can explain: the request that took 800 ms because for 700 of them nothing was running. It was waiting on a lock, a syscall, a channel, or the garbage collector. Execution traces exist for exactly this, but turning the tracer on in a long-lived service produces megabytes per second of data nobody will ever look at.

Go 1.25 solved this with runtime/trace.FlightRecorder: the tracer runs continuously, but instead of writing to disk it keeps a circular in-memory window of the last few seconds. When your code detects the anomaly, it dumps that window. It is exactly an aircraft's black box.

```
import (
"log"
"os"
"runtime/trace"
"sync"
"time"
)

var once sync.Once

func main() {
fr := trace.NewFlightRecorder(trace.FlightRecorderConfig{
// The window reliably retained. The official guidance is
// roughly 2x the size of the event you are investigating:
// for a 5 s timeout, use 10 s.
MinAge: 2 * time.Second,
// Memory cap. A busy service produces on the order of
// 10 MB/s of trace, so this limit is what governs.
MaxBytes: 16 << 20, // 16 MiB
})
if err := fr.Start(); err != nil {
log.Printf("flight recorder unavailable: %v", err)
}
defer fr.Stop()
```

```
// In the handler, after measuring latency:
// if fr.Enabled() && time.Since(start) > 500*time.Millisecond {
// go captureSnapshot(fr)
// }
}

func captureSnapshot(fr *trace.FlightRecorder) {
// once keeps a storm of slow requests from writing a
// thousand trace files.
once.Do(func() {
f, err := os.Create("/tmp/snapshot.trace")
if err != nil {
log.Printf("could not create snapshot: %v", err)
return
}
defer f.Close()
if _, err := fr.WriteTo(f); err != nil {
log.Printf("could not write snapshot: %v", err)
return
}
log.Printf("trace snapshot captured at %s", f.Name())
})
}
```

Open it afterwards with go tool trace /tmp/snapshot.trace. The "View trace by proc" view shows gaps: moments when every processor sat idle. Those gaps are the answer the CPU profile was never going to give you.

Profiles and traces don't compete: the profile answers "where was CPU spent," the trace answers "why wasn't it being spent."

Python: py-spy today, Tachyon tomorrow

Python's situation is different because the standard library profiler, cProfile, has no business in production. It is deterministic: it instruments every call. Beyond doubling or ten-folding runtime, it introduces a systematic bias, because instrumentation cost is proportional to the number of calls. Code with many small functions looks far more expensive than it is, and you end up "optimizing" something that costs nothing in production. Use cProfile in development and for short scripts; for production, use a sampling profiler.

py-spy: attaching to a process without touching it

py-spy is the de facto standard. It's written in Rust, runs outside the target process - reading its memory directly - and requires no code change and no restart. It is the tool you reach for at three in the morning when a worker has been pinned at 100% CPU for twenty minutes and nobody knows why.

```
# A live, top-style view of the hottest functions.
py-spy top --pid 4242

# Record for 60 seconds and produce an SVG flame graph.
py-spy record --pid 4242 --duration 60 --output flame.svg

# A Gunicorn/Celery worker with child processes: follow the whole tree.
py-spy record --pid 4242 --duration 60 --subprocesses --output flame.svg

# Only time where the thread actually holds the GIL. The difference
# between this and the normal profile is your I/O wait.
```

```
py-spy record --pid 4242 --duration 60 --gil --output gil.svg

# A single snapshot of every stack. This is what you run against a
# hung process: it tells you the exact line it is stuck on.
py-spy dump --pid 4242
```

The real obstacle is permissions, not the tool. py-spy reads another process's memory and that is restricted:

```
# Linux: Yama's ptrace_scope defaults to 1, which only allows attaching
# to child processes. If you see "Operation not permitted", this is why.
cat /proc/sys/kernel/yama/ptrace_scope
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope # temporary, until reboot

# Docker: the container needs the capability explicitly.
docker run --cap-add SYS_PTRACE myapp
```

In Kubernetes the pattern that works is a diagnostics container sharing the process namespace with the application, so the application itself needs no extra privileges:

```
apiVersion: v1
kind: Pod
metadata:
name: checkout-api
spec:
shareProcessNamespace: true # the sidecar can see the app's PIDs
containers:
- name: app
image: registry.example.com/checkout-api:1.4.2
- name: profiler
image: ghcr.io/benfred/py-spy:latest
command: ["sleep", "infinity"]
securityContext:
capabilities:
add: ["SYS_PTRACE"]
runAsUser: 0
resources:
limits:
cpu: 200m
memory: 128Mi
```

For memory, py-spy is the wrong tool: use memray, which tracks every allocation including those made by native C extensions, which is exactly where tracemalloc goes blind.

```
memray run --native -o output.bin -m myapp.worker
memray flamegraph output.bin # produces a browsable HTML report
memray attach 4242 # attach to an already-running process
```

What's coming in Python 3.15: profiling.sampling (Tachyon)

Python 3.15 finally brings a sampling profiler into the standard library. PEP 799 reorganizes the profilers into a profiling package with two modules: profiling.tracing (the familiar deterministic one) and profiling.sampling, code-named Tachyon. It builds on the safe external attach interface introduced by PEP 768, so like py-spy it reads the process from outside and, per the documentation, imposes no measurable overhead on the target.

```
# Attach to a live process for 30 s and write a self-contained flame graph.
python -m profiling.sampling attach --flamegraph -d 30 -o profile.html 12345
```

```
# Only time with the GIL held (modes are wall, cpu, gil and exception).
python -m profiling.sampling attach --mode=gil -d 30 12345

# Single snapshot of every thread's stack: a hung process.
python -m profiling.sampling dump --all-threads 12345

# asyncio applications: reconstruct the stack across await boundaries.
python -m profiling.sampling run --async-aware --async-mode=all worker.py

# Save a binary baseline and later produce a differential flame graph.
python -m profiling.sampling run --binary -o base.bin -m myapp
python -m profiling.sampling run --diff-flamegraph base.bin -o diff.html -m myapp
```

The default rate is 1 kHz, adjustable with -r (-r 10khz). Three limits are worth knowing before you plan anything around it:

- Profiler and target must run the same Python minor version. You cannot profile a 3.15 process from a 3.14 interpreter. With pre-release versions the requirement is stricter still: the exact same version. And you cannot cross between a free-threaded build and a standard one.
- Permission requirements match py-spy's: ptrace/process_vm_readv on Linux with CAP_SYS_PTRACE or ptrace_scope set to 0; task_for_pid() on macOS; administrative privileges or SeDebugPrivilege on Windows.
- --async-aware is incompatible with --native, --all-threads, and the cpu and gil modes, because it uses a different stack reconstruction mechanism.

In the meantime py-spy remains the sensible choice: it works with everything you already have deployed, including the Python versions that have been in production for years.

Node.js: --cpu-prof and an inspector session

In Node the quick path is a startup flag. When the process exits, V8 writes a .cpuprofile file you can open by dragging it into Chrome DevTools' Performance tab or into speedscope:

```
node --cpu-prof --cpu-prof-dir=/tmp/profiles server.js
```

That works for a script, not for a service that has been up for nine days and that you are not about to restart. For that, open an inspector session from inside the process itself, triggered by a signal or an admin endpoint:

```
const inspector = require('node:inspector');
const fs = require('node:fs/promises');
const path = require('node:path');

let running = false;

async function captureCpuProfile(seconds = 30) {
if (running) throw new Error('a capture is already in progress');
running = true;

const session = new inspector.Session();
session.connect();

const post = (method, params) =>
new Promise((resolve, reject) =>
session.post(method, params, (err, res) =>
err ? reject(err) : resolve(res)));
```

```
try {
await post('Profiler.enable');
// Sampling interval in microseconds. 1000 µs = 1 kHz.
await post('Profiler.setSamplingInterval', { interval: 1000 });
await post('Profiler.start');

await new Promise((r) => setTimeout(r, seconds * 1000));

const { profile } = await post('Profiler.stop');
const target = path.join('/tmp', 'cpu-' + Date.now() + '.cpuprofile');
await fs.writeFile(target, JSON.stringify(profile));
return target;
} finally {
session.disconnect();
running = false;
}
}

// Signal-triggered: kill -USR2 PID and you are done, no restart.
process.on('SIGUSR2', () => {
captureCpuProfile(30)
.then((f) => console.log({ msg: 'profile written', file: f }))
.catch((e) => console.error({ msg: 'profile failed', err: e.message }));
});
```

The running guard is not decorative: two concurrent V8 captures in the same process trample each other and the result is garbage.

For memory, require('node:v8').writeHeapSnapshot() exists and works, with one large caveat: it stops the world for the entire capture and the file is the size of the heap. On a process with a 3 GB heap that means several seconds of full pause and 3 GB of disk. It is a last-resort tool, never something you schedule every five minutes.

From ad-hoc profiling to continuous profiling

Everything above shares one flaw: you have to be there. The incident happens at 03:40, the pod restarts at 03:47, and by 09:00 when you open your laptop the process you wanted to profile no longer exists. Continuous profiling solves this the only way it can be solved: profile always, store the samples with labels, and let you query backwards in time.

The numbers that make this viable are modest. A typical sampling agent such as Grafana Pyroscope adds on the order of 2-5% CPU and under 50 MB of memory per pod, even with several profile types enabled at once. In exchange you get, for every deploy, the ability to ask "which function costs more CPU per request now than before?" and get an answer in thirty seconds.

Two ways to collect: SDK or eBPF

With an SDK you instrument the application. For one import and a few lines of configuration you get perfect symbols, business labels, and trace correlation. In Python:

```
import os
import pyroscope

pyroscope.configure(
application_name="checkout-api",
server_address=os.environ["PYROSCOPE_URL"],
# Count CPU time only; with False it measures wall clock
# (useful when your bottleneck is I/O rather than computation).
```

```
oncpu=True,
# These labels are what turn a profile into something actionable.
tags={
"env": os.environ["ENVIRONMENT"],
"region": os.environ["AWS_REGION"],
# The deploy SHA: without it you cannot attribute a
# regression to a specific change.
"version": os.environ["GIT_SHA"],
},
)

# Dynamic tagging per unit of work, inside a worker.
def process(job):
with pyroscope.tag_wrapper({"job_type": job.kind, "tenant": job.tenant_id}):
return run(job)
```

The version label gives more return per line of code than anything else here. With it, "p99 went up after yesterday's deploy" stops being a hypothesis: you filter the profile by the two SHAs, request the diff, and see the function that got fatter.

With eBPF you touch nothing. A privileged per-node agent samples the stacks of every process from the kernel. This is the option for heterogeneous fleets, third-party binaries, and teams that are not going to add a dependency to forty services. Configuration with Grafana Alloy running as a DaemonSet:

```
discovery.kubernetes "pods" {
role = "pod"
}

discovery.relabel "local_pods" {
targets = discovery.kubernetes.pods.targets

// Each agent profiles only the pods on ITS node. Without this rule,
// every agent tries to profile the entire cluster.
rule {
source_labels = ["__meta_kubernetes_pod_node_name"]
regex = env("HOSTNAME_NODE")
action = "keep"
}

rule {
source_labels = ["__meta_kubernetes_namespace", "__meta_kubernetes_pod_label_app"]
separator = "/"
target_label = "service_name"
}
}

pyroscope.ebpf "default" {
forward_to = [pyroscope.write.backend.receiver]
targets = discovery.relabel.local_pods.output
sample_rate = 97 // odd, to avoid locking with timers
collect_kernel_profile = false
}

pyroscope.write "backend" {
endpoint {
url = "http://pyroscope.observability.svc:4040"
}
external_labels = {
cluster = "prod-eu-west-1",
}
}
```

The price of eBPF is two concrete limitations worth understanding before you choose this path:

- Frame pointers. Reconstructing a stack from the kernel requires binaries compiled with frame

pointers. DWARF unwinding is not available inside the kernel, so if the binary and its libraries were built with -fomit-frame-pointer - the default behavior for two decades - you get truncated or outright useless stacks. The good news is that the industry reversed course: Fedora re-enabled them by default in version 38 and Ubuntu did the same in 24.04 LTS for 64-bit platforms. Go binaries always carry them. If your base image is an older distro or you compile C/C++ yourself, add -fno-omit-frame-pointer to your flags or the profile will be worthless.

- Interpreted and JIT languages. An eBPF profile of a Python process without a language-specific unwinder shows you _PyEval_EvalFrameDefault repeated forty times: accurate, and useless. You need an agent with an unwinder that understands the interpreter's internal structures. Modern agents ship them for Python, Java, Ruby, PHP, .NET, Node.js, and Perl, on top of native support for C/C++, Go, Rust, and Zig. Check your agent's support matrix against your fleet before promising full coverage.

The practical criterion: eBPF for broad, cheap coverage; SDK for the services you genuinely care about, where you want business labels and trace linking. They are not mutually exclusive.

Cardinality: the same mistake you already made with Prometheus

Every unique combination of labels is a separate profile series the backend must store and index. Add user_id as a label with a hundred thousand users and you have multiplied your storage cost by a hundred thousand. The rules are identical to metrics: low-cardinality, bounded labels (service, env, region, version, route_pattern), never free-form identifiers. The version SHA is the tolerable exception, because its cardinality grows with deploy rate and can be trimmed with retention.

The fourth signal: OpenTelemetry Profiles

Until now every profiling backend spoke its own dialect, and switching vendors meant swapping the agent across the whole fleet. OpenTelemetry is closing that gap: the profiles signal entered public alpha in March 2026, making profiling the fourth pillar alongside traces, metrics, and logs.

Two design decisions in the OTLP profiles format are worth calling out. The first is that it round-trips losslessly with pprof, the de facto format for a decade: you can convert in both directions without losing information, which means all of your existing tooling still applies. The second is a shared string dictionary that cuts wire size by roughly 40% - not a trivial saving once you multiply by thousands of pods.

The honest caveat: alpha means alpha. The OTLP specification is stable for traces, metrics, and logs, but still in development for profiles, and the SIG itself advises against relying on it for anything critical yet. The reasonable stance in 2026 is to treat it as an evaluation window: build your continuous profiling on what works today (Pyroscope, Parca, or your vendor's product) and trial the OTel signal in a non-critical environment so that migrating, once it reaches beta and GA, is an exporter swap rather than a project.

Linking traces and profiles today: span profiles

What you can have right now is the direct jump from a slow span to the stacks that ran during it. The mechanism is simple: when a span starts, profile samples are tagged with its span_id. With that, the trace UI can render a flame graph for that specific span.

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from pyroscope.otel import PyroscopeSpanProcessor

provider = TracerProvider()
provider.add_span_processor(BatchSpanProcessor(OTLPSpanExporter()))
# Tags profile samples with the active span.
provider.add_span_processor(PyroscopeSpanProcessor())
trace.set_tracer_provider(provider)
```

The operational result is the scenario this guide opened with, but resolved: you see a 640 ms render_cart span, click the profile tab, and find that 70% of it goes into a regex compiled inside a loop. That is the difference between knowing something is slow and knowing what to fix.

Memory: when the heap profile says nothing

There is one scenario that baffles a lot of people. The pod dies with OOMKilled against a 2 GiB limit. You open the heap profile and it sums to 900 MB. The numbers don't add up and the hasty conclusion is that the profiler is lying. It isn't: it is measuring something else.

A process's RSS is the runtime heap plus several things no managed heap profile will ever show you:

- Fragmentation and lazily returned memory. The allocator reserves from the OS in large chunks and doesn't always give them back when an object dies. In Go, runtime/metrics and go_memstats_heap_released_bytes give you the real picture; in glibc, the number of per-thread arenas can inflate RSS dramatically in heavily threaded processes, and MALLOC_ARENA_MAX=2 is a classic fix.
- Native allocations. A buffer allocated inside a C extension - NumPy, a database client, an image codec - never goes through the language allocator and never appears in the profile. In Python this is exactly what separates tracemalloc (only sees interpreter allocations) from memray --native (sees all of them).
- Thread stacks, file mappings, the binary itself and its libraries. Ten thousand goroutines with 8 KiB of initial stack is 80 MiB that is not in the heap profile.

The diagnostic sequence that works: first compare RSS (from container_memory_working_set_bytes, not from free) against the live heap the runtime reports. If the heap grows, you have an object leak and the inuse_space profile will point straight at it. If the heap is flat while RSS climbs, the problem is fragmentation or native allocations, and there you need a profiler that goes down to the system allocator level.

In Go there is also one lever that resolves many OOMKills without changing a line of logic: GOMEMLIMIT. Setting it to around 90% of the container limit makes the collector work harder as it approaches the ceiling rather than letting the kernel kill the process. It does not fix a real leak, but it converts a transient spike into a little extra CPU instead of an outage.

The JVM: JFR and async-profiler, or why your Java flame graph was lying

If your platform is Java or Kotlin, the Go chapter doesn't transfer as-is, and not because of syntax: JVM

profilers have a problem that Go and Python profilers don't. Most of the classic tools - VisualVM, JProfiler in sampling mode, anything built on ThreadMXBean.getThreadInfo() - can only capture a thread's stack when that thread is at a safepoint. And safepoints are not spread evenly through your code: the JIT places them at loop back-edges, method calls and allocations, but aggressively removes them from counted loops over arrays and from intrinsic code. The result has a name: safepoint bias. The profiler doesn't show you where the CPU is, it shows you where the nearest safepoints to the CPU are. A numeric loop burning 40% of the process can show up attributed to its caller, or not show up at all.

The practical consequence is uncomfortable: plenty of people have optimised the wrong function for years using data that looked correct because the flame graph rendered nicely. A pretty flame graph is not a true flame graph.

async-profiler: sampling without safepoint bias

async-profiler solves this with AsyncGetCallTrace, an internal HotSpot API that can unwind a Java thread's stack from inside a signal handler, without waiting for a safepoint. The signal comes from the kernel's performance events subsystem (perf_events) or a POSIX timer, so sampling really is proportional to CPU time consumed. It also unwinds the native stack, so you see JIT-compiled code, the interpreter, JNI calls and the JVM runtime itself in the same graph: when 30% of your time is in G1ParScanThreadState or jbyte_disjoint_arraycopy, you see it, and no Java-only profiler would have shown you that.

Version 4.3, released in January 2026, is the first of the year and consolidated two things that matter in production: native JFR output and improved wall-clock sampling. The project has settled into a release every three months, which in practice means you can pin a version and upgrade quarterly instead of chasing master.

```
# Attach to a running process by PID, 30 seconds of CPU, HTML output
./asprof -d 30 -e cpu -f /tmp/cpu-profile.html $(pgrep -f my-service.jar)

# Wall clock: includes blocked time, not just CPU time.
# Essential when latency rises but CPU doesn't.
./asprof -d 30 -e wall -t -f /tmp/wall-profile.html <pid>

# Lock contention and allocations, the two profiles that solve the most
# problems in Java services after the CPU one
./asprof -d 30 -e lock -f /tmp/lock-profile.jfr <pid>
./asprof -d 30 -e alloc -f /tmp/alloc-profile.jfr <pid>

# As an agent, from startup, recording continuously into JFR.
# --wall adds wall-clock sampling at its own coarser interval, so a single
# recording contains both on-CPU and off-CPU time.
java
-agentpath:/opt/async-profiler/lib/libasyncProfiler.so=start,event=cpu,--wall=50ms,jfr,file=/var/log/
perf-%p-%t.jfr,loop=30m \
-jar my-service.jar
```

Two operational details that cost an afternoon if you don't know them. First: stack unwinding needs the JVM started with -XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints, because without it the JIT's debug map only has entries at safepoints and you are back, by another route, to the problem you were trying to avoid. Second: inside a container, attaching to a process requires the same PID namespace and the SYS_PTRACE capability, exactly as with py-spy; if your plan is on-demand profiling in Kubernetes, either bake the agent into the image or use an ephemeral container with shareProcessNamespace: true.

JFR: the one you already have installed and almost nobody turns on

JDK Flight Recorder has shipped in OpenJDK since Java 11 (and was backported to 8u), it lives in the JDK itself, and its default configuration has been engineered since the JRockit days to stay under 1% overhead. That target has held release after release: on the SPECjbb2015 reference workload the default configuration consistently measures at around 1% or less. There are 500-plus built-in event types - GC, JIT, threads, sockets, file I/O, exceptions, class loading, heap statistics - and unlike an external profiler, you don't have to install anything or attach to anything.

What JFR historically lacked was precisely an honest CPU profile: its ExecutionSample event carried the same safepoint bias. That changed in the JDK 25 LTS cycle, where JEP 509 introduced JFR CPU-time profiling based on the kernel's CPU clock on Linux - conceptually the same approach as async-profiler, but inside the JDK and with no third-party agents. If you are on JDK 25 or later on Linux, the "I can't install a .so in production" argument is gone.

```
# Continuous ring recording: you always have the last 15 minutes in memory
# and a dump to disk when something goes wrong. Cost < 1%.
java -XX:StartFlightRecording=settings=default,maxage=15m,maxsize=200m,name=cont \
-XX:FlightRecorderOptions=stackdepth=128 \
-jar my-service.jar

# Dump the ring NOW, without restarting anything, when the alert fires
jcmd $(pgrep -f my-service.jar) JFR.dump name=cont filename=/tmp/incident.jfr

# Read it without opening JMC: the JDK ships a command-line reader
jfr summary /tmp/incident.jfr
jfr print --events ExecutionSample --stack-depth 16 /tmp/incident.jfr | head -60
jfr print --events JavaMonitorEnter,ThreadPark /tmp/incident.jfr | head -40
```

The combination that works best in practice isn't choosing between the two, it's using each for what it's good at: JFR on all the time, in ring mode, as the process's black box - it gives you full context for the minute before an incident, including GC pauses and I/O events that no sampling profiler records - and async-profiler on demand when you need a fine-grained CPU or allocation flame graph. One versioning detail: the timeSpan field that 4.3 added to the profiler.WallClockSample event makes older JFR converters fail on new recordings; the new converter does read old ones, so the rule is upgrade the reader first and the agent second.Off-CPU: the profile that explains the 800 ms the CPU never spent

Off-CPU: the profile that explains the 800 ms the CPU never spent

There is an entire class of latency problems the CPU profile is blind to by construction, and it is the most common class in network services. Your trace says the span took 800 ms. You open the CPU flame graph for those 800 ms and find 12 ms scattered as noise. There is no contradiction and no bug: the process wasn't executing anything. It was waiting - on a database query, on a lock, on a disk write, on the scheduler because the cgroup exhausted its CPU quota - and a profiler that samples the CPU cannot sample anyone who isn't on the CPU.

The profile that answers that question is called off-CPU and it is built the other way round: instead of interrupting periodically, it hooks the point in the kernel scheduler where a thread leaves the CPU, records the stack and the timestamp, and when that same thread runs again computes how long it was away. The result isn't "how many samples landed here" but "how many microseconds blocked here", and that directly measures I/O waits, lock contention and everything the scheduler counts as sleep.

```
# bcc ships the tool. Installation:
# Debian/Ubuntu: apt-get install bpfcc-tools
# RHEL/CentOS: yum install bcc-tools

# 30 seconds of off-CPU time for one PID, as folded stacks, with user and
# kernel stacks. The -m threshold drops trivial blocking.
sudo /usr/share/bcc/tools/offcputime -df -p $(pgrep -f my-service) -m 1000 30 \
> /tmp/off.folded

# Convert to a flame graph (Brendan Gregg's FlameGraph)
./flamegraph.pl --colors=io --title="Off-CPU 30s" /tmp/off.folded > /tmp/off.svg
```

Three things worth knowing before you read the result. First: the X axis of an off-CPU flame graph is microseconds blocked, not samples, so an idle thread that bothers nobody - a connection pool waiting for work, a timer - will dominate the graph and mean nothing. That's why you almost always want to filter by thread or duration range rather than stare at the aggregate. Second: offcputime charges per event, not per interval, and a service with hundreds of thousands of context switches per second will notice; the -m threshold and a specific -p are what keep it cheap. Third: if the process is blocked in a syscall and you have no frame pointers in libc, the stack truncates right where it starts getting interesting - which brings us to the next section.

The wall-clock profile: adding both into one graph

What you actually want to see, nearly always, is the sum: wall clock = on-CPU + off-CPU. If you capture both over the same window and normalise the scales - CPU samples at 99 Hz represent roughly 10.1 ms each; off-CPU already comes in microseconds - you can merge the folded stacks and get a single flame graph where a branch's width is real elapsed time. It's the view that turns "the service is slow" into "68% of this request's time is spent waiting on Postgres from this method".

```
#!/usr/bin/env python3
"""Merge on-CPU and off-CPU folded stacks onto a common microsecond scale.

on.folded : output of 'profile -f' (samples at HZ hertz)
off.folded : output of 'offcputime -df' (microseconds blocked)
"""
import sys
from collections import defaultdict

HZ = 99 # sampling frequency used for the on-CPU profile
US_PER_SAMPLE = 1_000_000 / HZ

def load(path, factor):
total = defaultdict(float)
with open(path) as fh:
for line in fh:
line = line.strip()
if not line:
continue
stack, _, value = line.rpartition(" ")
if not stack:
continue
total[stack] += float(value) * factor
return total

def main(on_path, off_path):
combined = defaultdict(float)
# Tag each branch so the two can be coloured differently in the flame graph
for stack, us in load(on_path, US_PER_SAMPLE).items():
combined[stack + ";[on-cpu]"] += us
for stack, us in load(off_path, 1.0).items():
combined[stack + ";[off-cpu]"] += us
```

```
total = sum(combined.values()) or 1.0
for stack, us in sorted(combined.items(), key=lambda kv: -kv[1]):
# flamegraph.pl expects integers; round to microseconds
sys.stdout.write(f"{stack} {int(us)}\n")
print(f"# total = {total/1e6:.2f} s of wall clock", file=sys.stderr)

if __name__ == "__main__":
main(sys.argv[1], sys.argv[2])
```

With that in place, diagnosis changes character. A wide plateau tagged [off-cpu] under a psycopg call isn't "the database is slow": it's a concrete number of wall-clock seconds you can line up against pg_stat_statements on the other side. And a wide [off-cpu] plateau under futex_wait with your own code on both sides is lock contention, which is your problem and nobody else's.

In specific languages there are shortcuts that avoid assembling all of this. In Go, the block and mutex profiles we already covered handle the synchronisation part of off-CPU without leaving the process. In Java, asprof -e wall -t gives you a per-thread wall-clock profile directly. In Python, py-spy record --idle includes threads that are waiting, which are dropped by default - and that single flag is the difference between a flame graph that explains your latency and one that insists your service does nothing.Symbols: why your flame graph is full of hexadecimal addresses

Symbols: why your flame graph is full of hexadecimal addresses

The first flame graph most people get out of a system profiler is a disappointment: a tower of bars reading 0x7f3a91c4e210 and not much else. It isn't a misconfiguration; it's that native profiling has two distinct problems that are solved separately, and conflating them wastes days. The first is unwinding the stack: working out the chain of return addresses. The second is symbolising: translating each address into a function name, a file and a line.

Unwinding: frame pointers versus DWARF

The cheap way to unwind is the frame pointer. If the compiler reserves rbp to chain frames, walking the stack is following a linked list, and that is exactly what the kernel helper bpf_get_stackid does: tens of nanoseconds, inside signal context, without touching arbitrary user memory. The problem is that for fifteen years the standard practice was to compile with -fomit-frame-pointer to reclaim a register, and that register was worth roughly 1% of performance. In exchange, the whole ecosystem lost the ability to profile itself.

The alternative is DWARF. ELF binaries carry an .eh_frame section - created originally for C++ exception unwinding and later repurposed as a general format - that describes, for every address in the program, how to reconstruct the caller's frame. It is complete information and it is expensive information: .eh_frame contains a bytecode interpreter, something you cannot run inside an eBPF program with its instruction limit and no unbounded loops. The solution modern profilers adopted is to precompute: read .eh_frame in user space, flatten it into a table of stack deltas - for each address range, which register is the base and what offset holds the return address - and load that table into eBPF maps so the kernel-side unwinder only has to do binary searches.

The good news is that the industry reversed course. Fedora 38 and Ubuntu 24.04 re-enabled frame pointers in their default packages, and from there the fast path works again for most distribution software. The practical rule for your own binaries is simple: in Go you already have them; in Rust and

C++ turn them on explicitly and accept the 1%.

```
# Rust: frame pointers in the release profile, without touching Cargo.toml
RUSTFLAGS="-C force-frame-pointers=yes" cargo build --release

# C/C++
cc -O2 -g -fno-omit-frame-pointer -fasynchronous-unwind-tables ...

# Check that a binary carries what's needed to unwind
readelf -S ./my-binary | grep -E 'eh_frame|debug_'
# And if the prologue starts with 'push %rbp; mov %rsp,%rbp', frame pointers are on:
objdump -d ./my-binary | grep -A2 '<main>:' | head -3
```

Symbolising: where the names live and why they aren't where you profile

Once the stack is unwound, you need the names. And here the universal container-image practice bites: strip. A well-built production image carries no symbol table, because it's heavy and it isn't needed to run. The profiler, running as a DaemonSet on the node, finds the binary but not the names.

There are three ways out, and picking the right one depends on your build pipeline. The first is symbolise at the source: the agent uploads unresolved addresses to the backend and the backend resolves them against a symbol store you populated with debug files at build time. That's what Polar Signals and Parca do, and it's the sane option if you control CI. The second is debuginfod: a simple HTTP protocol that serves debug files indexed by Build ID, the unique identifier the linker leaves in each binary's .note.gnu.build-id section. Fedora, Debian and Ubuntu run public servers for their packages, and standing one up for your own artifacts is two hours of work. The third, the worst, is leaving symbols inside the image and paying the size.

```
# At build time: split symbols into a separate file and leave a Build ID link
# inside the binary. The image stays small and symbols remain recoverable.
objcopy --only-keep-debug my-binary my-binary.debug
objcopy --strip-debug my-binary
objcopy --add-gnu-debuglink=my-binary.debug my-binary

# The Build ID is the key you recover them with later
readelf -n my-binary | grep -A1 'Build ID'
# -> Build ID: 4a7f1c9e2b5d8a3f6c0e9b1d4a7f1c9e2b5d8a3f

# Publish into a store indexed by Build ID (debuginfod layout)
BID=$(readelf -n my-binary | awk '/Build ID/{print $3}')
install -D my-binary.debug "/srv/debuginfo/${BID:0:2}/${BID:2}.debug"

# On the machine where you symbolise
export DEBUGINFOD_URLS="https://debuginfod.internal/ https://debuginfod.elfutils.org/"
debuginfod-find debuginfo "$BID" # downloads and caches in ~/.cache/debuginfod_client
```

Languages with a runtime have their own set of traps. Go embeds its symbol table in the binary, so it almost always works with no effort - unless someone put -ldflags="-s -w" in the Dockerfile to save six megabytes, which is the number one cause of unreadable Go profiles. Interpreted languages are a different and deeper problem: the native stack of a Python process shows _PyEval_EvalFrameDefault twenty times over, because Python functions are not native frames. That's why system profilers need per-runtime unwinders that read the interpreter's internal structures to reconstruct the language-level stack. OpenTelemetry's eBPF profiler, donated by Elastic and now an official Collector component, ships runtime support for Go, Node.js (V8), Ruby, .NET 9 and 10, and initial support for BEAM (Erlang and Elixir). Each of those unwinders depends on internal offsets specific to the runtime version, which explains the rule that looks arbitrary and isn't: if you upgrade the interpreter, upgrade the profiler, or the

symbols will quietly turn to garbage.Profiling in CI: turning a CPU regression into a failed build

Profiling in CI: turning a CPU regression into a failed build

Continuous profiling tells you a regression reached production. Profiling in CI keeps it from getting there. They are different things and both are worth doing, but the second is almost always built badly, because people try to put an absolute threshold ("this benchmark must run in under 5 ms") on shared runners whose performance varies 30% depending on the neighbour. The result is a test that fails on Tuesdays and ends up marked continue-on-error, which is an elegant way of deleting it.

The form that works has three rules. One: compare, don't measure. Run the base branch and the PR branch in the same job, on the same machine, interleaved; what matters is the relative difference, and that one is stable. Two: repetitions and statistics, not a single number. benchstat applies a Mann-Whitney test and only reports a change when it is significant; one run cannot tell an 8% regression from noise. Three: keep the profile, not just the number. A build that fails saying "+14% CPU" is useful; one that also attaches the differential pprof pointing at the responsible function is what gets it fixed that same afternoon instead of filed as a ticket.

```
#!/usr/bin/env bash
# ci/perf-gate.sh - fail the build if a benchmark regresses significantly
set -euo pipefail

BASE="${1:-origin/main}"
PKG="${2:-./internal/pipeline}"
N=10 # repetitions per side
THRESHOLD_PCT=5 # tolerance

work=$(mktemp -d); trap 'rm -rf "$work"' EXIT

measure () { # $1 = label
go test "$PKG" -run '^$' -bench . -benchmem \
-count "$N" -cpuprofile "$work/$1.pprof" \
| tee "$work/$1.txt"
}

git stash push --include-untracked --quiet || true
git checkout --quiet "$BASE"
measure base
git checkout --quiet -
git stash pop --quiet || true
measure pr

# Statistical comparison: benchstat only reports significant differences
benchstat "$work/base.txt" "$work/pr.txt" | tee "$work/benchstat.txt"

# Differential profile: which function ate the extra time
go tool pprof -top -nodecount=15 \
-base "$work/base.pprof" "$work/pr.pprof" \
| tee "$work/diff.txt"

worst=$(awk '/sec\/op/ {if (match($0, /\+([0-9.]+)%/, m) && m[1]+0 > max) max=m[1]+0} END {print
max+0}' \
"$work/benchstat.txt")

if (( $(echo "$worst > $THRESHOLD_PCT" | bc -l) )); then
echo "::error::Performance regression: +${worst}% (threshold ${THRESHOLD_PCT}%)"
cp "$work"/{benchstat.txt,diff.txt,pr.pprof,base.pprof} "${ARTIFACTS_DIR:-.}/"
exit 1
fi
echo "No significant regression (worst case: +${worst}%)"
```

pprof's -base flag is the piece almost nobody uses and the one that saves the most time: it subtracts one profile from another and leaves you only the delta, so instead of eyeballing two flame graphs - a task the human eye is remarkably bad at - you see directly the five functions that appeared or grew. There is also -diff_base, which normalises by the total before subtracting; use -base when you want absolute growth and -diff_base when total load changed between the two runs.

One warning so you don't build a religion around this: a performance test in CI covers the hot path you chose to measure, with the data you generated. A regression that only shows up with production's real payload size distribution will not appear here. Continuous profiling and CI profiling don't compete; the second catches the obvious regressions and the first finds the ones that aren't.

What it costs, and how not to pay for it twice

Continuous profiling has two costs and almost everyone budgets for one. The cost inside the profiled process we've already seen and it is small: 2-5% of CPU with an eBPF agent, under 1% with JFR in its default configuration, a few tens of megabytes of memory per pod. The second cost is the backend, and that one scales with decisions you make without noticing.

The shape of managed-service pricing explains it well. Grafana Cloud Profiles, for instance, bills three separate line items - processing per gigabyte, writes per gigabyte, and retention per gigabyte per month - with the free plan around 50 GB a month and 14 days of retention, and the paid plan at 30 days. Real deployments overshoot the initial estimate with monotonous regularity, and the cause is nearly always the same one as in Prometheus: cardinality. Every distinct combination of labels is an independent profile series that is stored, indexed and retained separately. Adding version with the deployment SHA is one of the best decisions you can make, because it turns a regression into something attributable; adding request_id is the same mistake you already made once with metrics, except profiles weigh three orders of magnitude more than a counter.

The four levers that actually move the bill, in order of effect: which services you profile - the common practice in large organisations is not to profile everything but the expensive services and the ones on the critical path; a CronJob that runs three minutes a day does not need continuous profiling; the sampling frequency, where dropping from 99 Hz to 19 Hz cuts volume almost fivefold and remains perfectly usable for multi-day trends, though no longer for pinpointing a single function; tiered retention, keeping high resolution for two weeks and low-resolution aggregates for a year, which is what you need to answer "when did this start climbing?"; and the number of active profile types, because enabling CPU, heap, allocations, goroutines, mutex and block at once multiplies ingest sixfold when in practice CPU and heap answer 80% of the questions.

On the self-hosted side, Pyroscope 2.0 - released on 21 April 2026 - rebuilt precisely the storage and query layer that had become expensive and operationally awkward in the original design, so if you evaluated Pyroscope a year ago and dropped it over operational cost, that evaluation is stale. And if your criterion is avoiding vendor lock-in, the reasonable entry point in 2026 is the format: OTLP profiles round-trips losslessly against pprof, so the data you collect today with an SDK stays yours and stays convertible tomorrow.A full case: the 38% of CPU nobody asked for

A full case: the 38% of CPU nobody asked for

It's worth seeing how the pieces fit on a real problem, because in an incident you never use one tool alone. What follows is a composite but representative case: an ingestion service in Go, twelve replicas,

around 4,000 requests per second, which after an upgrade started needing sixteen replicas for the same traffic. Nothing was down. The p99 latency had gone from 82 ms to 104 ms, comfortably inside the SLO. The compute bill had risen 33%.

What the metrics said. CPU per replica at 71% against 48% before. Requests per second identical. No change in payload size. Go's GC running more often, but with normal pauses. The metrics bounded the problem - it's CPU, it isn't I/O, it isn't a dependency - and there they ran out, which is exactly what you should expect from them.

What the traces said. The handle_batch span had gone from 31 ms to 44 ms on average. Inside it there were no instrumented children. Thirteen milliseconds appearing inside a span with no internal structure: the trace had done its job and could do no more.

What the profile said. With Pyroscope already collecting profiles labelled by version, the query was direct: CPU profile of the current deployment minus CPU profile of the previous one, same time window on the same weekday. The differential pointed at a function nobody had touched, encoding/json.Marshal, at 38% of CPU time against 11% before. Walking up the stack, the new caller was a validation function added in the upgrade which, to check that an optional field matched a schema, serialised the entire struct to JSON - including a metadata field that could run to tens of kilobytes - and then threw it away.

```
# What was actually run, reduced to essentials. Both profiles are pulled from
# the backend by version label; -base leaves only the delta.
pprof-cli download --service ingest --profile cpu \
--from '2026-09-08T14:00Z' --to '2026-09-08T14:30Z' \
--selector 'version="v2.14.0"' -o previous.pprof
pprof-cli download --service ingest --profile cpu \
--from '2026-09-15T14:00Z' --to '2026-09-15T14:30Z' \
--selector 'version="v2.15.0"' -o current.pprof

go tool pprof -top -nodecount=12 -base previous.pprof current.pprof
# flat flat% sum% cum cum%
# 4.21s 38.1% 38.1% 5.02s 45.4% encoding/json.(*encodeState).marshal
# 0.63s 5.7% 43.8% 5.65s 51.1% ingest/validate.CheckOptionalSchema
# 0.29s 2.6% 46.4% 0.29s 2.6% runtime.mallocgc
# -0.11s -1.0% 45.4% -0.11s -1.0% ingest/decode.Fast
```

The fix was four lines: check the field is present before validating it, and validate only the relevant subtree instead of the whole struct. But the interesting part of the case isn't the fix, it's the timeline. Without profiles, this would have been resolved - it was resolved this way in the same team a year earlier, on an analogous problem - by adding replicas, filing an "investigate CPU usage" ticket, and finding it three weeks later while looking at something else. With profiles and the version label, from the cost alert to the guilty line took eleven minutes, and nine of them were waiting for someone with access to open the dashboard.

There are two operational lessons in there, and neither is about JSON. First: the label carrying the deployment SHA or version is what turns a profile into an instrument of attribution; without it you would have seen that json.Marshal is expensive - which is true in almost any Go service and tells you nothing - instead of seeing that it grew. Second: the problem never fired a latency alert, because it never broke the SLO. The compute budget caught it. A meaningful share of the regressions continuous profiling finds are of this kind: they break nothing, they just cost money forever, and they are invisible to an observability system that only watches error and latency thresholds.

Eight recurring mistakes

1. Exposing pprof on the public mux. Covered above, but worth repeating because it remains the most frequent failure: import _ "net/http/pprof" plus http.ListenAndServe(":8080", nil) is an internet-reachable memory dump and denial of service.
2. Profiling for five seconds and drawing conclusions. With few samples, a 5% figure could be 3% or 7%. If your "optimization" moves the needle by less than 2%, you have measured nothing.
3. Reading a flame graph's X axis as time. It isn't. For temporal sequence you need an execution trace.
4. Confusing alloc_space with inuse_space. Optimizing the function that has allocated the most since startup when what you have is a leak is wasted time with a feeling of progress.
5. Leaving mutex and block profiling off. Those are Go's defaults, and they mean that contention regressions - a large share of the total - are invisible to you.
6. Running eBPF profiling without frame pointers. You get truncated stacks and lose a week blaming the tool. Verify your base image's compile flags before deploying the agent.
7. Putting identifiers in labels. user_id, request_id, or a URL with IDs embedded do to a profiling backend exactly what they do to Prometheus: melt it.
8. Comparing two windows with different load. A diff between 04:00 and 12:00 shows that everything grew. That isn't a regression, it's traffic. Normalize per unit of work or compare equivalent windows.

Production checklist

- pprof endpoints (or your language's equivalent) listen on loopback or on an admin port that is not in the Service or the Ingress.
- The admin listener's WriteTimeout exceeds the longest profile duration you intend to request.
- In Go, SetMutexProfileFraction and SetBlockProfileRate are enabled with conservative values, measured under load before going to production.
- Base images are compiled with -fno-omit-frame-pointer if you plan to use an eBPF agent.
- Every sample carries at minimum the service, env, region, and version (commit SHA) labels.
- No label contains user IDs, request IDs, or unnormalized URLs.
- There is an explicit retention policy: high resolution for days, aggregates for weeks or months.
- Agent overhead is measured against your real workload rather than assumed from the README, and there is an alert if the agent stops reporting.
- The incident runbook includes the step "open the differential flame graph against the previous deploy" before "roll back blindly".
- Profiles are treated as potentially sensitive data: function and file names reveal internal structure, and in some cases arguments. Access control accordingly.

FAQ

What does leaving this on all the time actually cost? With a modern sampling agent, on the order of 2-5% CPU and under 50 MB of memory per pod. The dominant cost isn't the agent, it's storage, and you

control that with retention and cardinality discipline.

Can I profile only when there's a problem? You can, and it beats nothing, but you lose the most valuable part: the baseline. Without history you have nothing to compare against, and an isolated flame graph tells you where your service spends CPU, not what changed.

Profile or execution trace? The profile answers "where was CPU spent." The execution trace answers "why wasn't it being spent": waits, blocks, GC pauses, scheduling. If your p99 is high but CPU is idle, your answer is in the trace, not the profile.

Is this worth it for an I/O-bound service? Yes, but change the mode. A CPU profile of a service that spends 95% of its time waiting on Postgres shows very little. Profile in wall-clock mode (oncpu=False in Pyroscope, --mode=wall in Tachyon, py-spy's default) and you'll see where the waiting happens.

What if my language isn't on the list? Almost any runtime with perf support can feed a flame graph through the collapsed stacks format, which is one line of text per stack plus a counter. It is the lowest common denominator of the entire ecosystem.

Should I wait for OpenTelemetry Profiles to reach GA? No. Build continuous profiling now with what works and evaluate the OTel signal in parallel. When it matures, migrating will be an exporter swap; what you can never recover is the year of history you didn't collect.

Glossary

- Sampling profiler - interrupts or inspects the program at fixed intervals and records the stack. Cost is proportional to frequency, not to work done.
- Deterministic (tracing) profiler - instruments every call. Exact on counts, expensive, and biased toward code with many small calls.
- Flame graph - aggregated stack visualization: width equals samples, height equals stack depth, X axis carries no temporal meaning.
- Self time (flat) - time in a function excluding its callees. This is what shows up as a plateau at the top of a flame graph.
- Cumulative time (cum) - time in a function including everything it calls.
- pprof - Google's profile format (protobuf) and the tool that reads it. The industry's de facto standard.
- Collapsed stacks - a text format with one semicolon-separated stack per line followed by a sample count. The universal interchange between tools.
- Frame pointer - a register pointing at the previous stack frame. Without it, unwinding the stack from the kernel is impractical and the profile comes out truncated.
- eBPF - the Linux kernel virtual machine that runs verified programs at kernel hook points. The basis of profilers that need no in-process agent.
- Span profile - profile samples tagged with the active span's identifier, letting you jump from a slow trace to its stacks.
- Flight recorder - a tracer that keeps a recent circular window in memory and dumps it only when the program detects an anomaly.

Where to start

If you have nothing set up, this order produces results fastest. First, expose pprof (or your language's equivalent) on an admin port in a single service - the one consuming the most CPU on your bill. Second, capture a 60-second baseline and store it in the repository alongside the version. Third, deploy a continuous agent on that same service with four labels: service, env, region, and version. Fourth, the next time you deploy, open the diff against the previous SHA.

That fourth step is what convinces the rest of the team, because it's the first time anyone can point at a specific function and say "this cost 12 ms per request yesterday and it costs 40 today." Everything else - fleet-wide coverage, trace linking, the OpenTelemetry migration - comes afterwards and is much easier to justify.