

Logging Estructurado: JSON, Contexto y Correlación con Trazas en Producción

Edición en español

Convierte tus logs en datos consultables: salida JSON con structlog (Python) y pino (Node.js), contexto por petición con request_id, redacción de datos sensibles, correlación con trazas de OpenTelemetry y los errores que hacen inútiles tus logs justo cuando más los necesitas. Edición ampliada: ProcessorFormatter para unificar bibliotecas de terceros, excepciones con contexto, canonical log lines, logging en workers y colas, exportación OTLP, muestreo y control de costes, debug dirigido, tormentas de logs, esquema común (OTel/ECS), Kubernetes por dentro, Fluent Bit y Vector, buffering y backpressure, serverless, multi-tenant, seguridad y cumplimiento, testing de logs, consultas LogQL y CloudWatch, migración de servicios legacy, un caso práctico completo de incidente y FAQ. Con código listo para producción.

Guía · Intermedio · 45 min de lectura · Julio 2026

Cuando algo falla a las tres de la mañana, la diferencia entre resolver el incidente en diez minutos o en dos horas suele estar en los logs. Si son líneas de texto libre, te espera una sesión de grep y expresiones regulares. Si son eventos JSON con campos consistentes, una consulta te da la respuesta. Esta guía cubre cómo montar logging estructurado de verdad: configuración de structlog y pino, contexto por petición, redacción de datos sensibles, correlación con trazas de OpenTelemetry, canonical log lines, logging en workers y colas, exportación por OTLP, control de costes, envío con Fluent Bit y Vector, testing de logs y un caso práctico completo de investigación de un incidente.

De texto libre a eventos consultables

Un log tradicional es una frase pensada para humanos: "Usuario 42 creó el pedido ord_123 por 49,99 €". Para encontrarla necesitas saber exactamente cómo se redactó. Un log estructurado es un evento con datos:

```
{
  "timestamp": "2026-07-21T09:14:03.512Z",
  "level": "info",
  "event": "order_created",
  "service": "checkout-api",
  "user_id": 42,
  "order_id": "ord_123",
  "amount_cents": 4999,
  "currency": "EUR",
  "request_id": "3f2a91c0-7b2e-4d1a-9c55-8f0f2b6a1e77"
}
```

La diferencia práctica es enorme. Con eventos estructurados puedes preguntar "todos los pedidos fallidos del usuario 42 en la última hora" o "errores agrupados por endpoint" directamente en tu plataforma de logs (Loki, Elasticsearch, Datadog, CloudWatch), sin regex frágiles. El mensaje deja de ser prosa y pasa a ser un nombre de evento estable —order_created, payment_failed— que puedes contar, agrupar y usar en alertas.

Los campos que ningún log debería omitir

- timestamp en ISO 8601 y UTC. Mezclar zonas horarias entre servicios convierte cualquier investigación en arqueología.
- level (debug, info, warning, error): filtrar por severidad es lo primero que harás en un incidente.
- event: un nombre corto y estable en snake_case, no una frase. "payment_failed" se puede agregar; "el pago del usuario X falló por..." no.
- service y versión desplegada, para saber qué código produjo el log.
- request_id / trace_id: el identificador que une todos los logs de una misma petición a través de tus servicios. Sin él, correlacionar es adivinar.
- Contexto del dominio: user_id, order_id, job_id... los identificadores por los que buscarás.

Python: structlog bien configurado

structlog es el estándar de facto para logging estructurado en Python. Una configuración de producción razonable:

```
import logging
import structlog

structlog.configure(
    processors=[
        structlog.contextvars.merge_contextvars,
        structlog.processors.add_log_level,
        structlog.processors.TimeStamper(fmt="iso", utc=True),
        structlog.processors.format_exc_info,
        structlog.processors.JSONRenderer(),
    ]
)
```

```

    ],
    wrapper_class=structlog.make_filtering_bound_logger(logging.INFO),
    cache_logger_on_first_use=True,
)

log = structlog.get_logger()
log.info("order_created", order_id="ord_123", amount_cents=4999, currency="EUR")
# {"order_id": "ord_123", "amount_cents": 4999, "currency": "EUR",
#  "event": "order_created", "level": "info", "timestamp": "2026-07-21T09:14:03.512512Z"}

```

Cada procesador hace una sola cosa: `merge_contextvars` incorpora el contexto de la petición actual (lo vemos ahora), `TimeStamper` añade la marca de tiempo en UTC, `format_exc_info` serializa la excepción cuando llamas a `log.exception()`, y `JSONRenderer` emite la línea final. En desarrollo puedes sustituir este último por `structlog.dev.ConsoleRenderer()` y tener salida legible con colores sin tocar nada más.

Contexto por petición sin pasar el logger a mano

El error clásico es intentar arrastrar `request_id` como argumento por todas las capas de la aplicación. Los `contextvars` lo resuelven: enlazas los datos una vez en un middleware y aparecen en todos los logs de esa petición, incluso dentro de código `async`.

```

import uuid
from structlog.contextvars import bind_contextvars, clear_contextvars

@app.middleware("http")
async def logging_context(request, call_next):
    clear_contextvars()
    request_id = request.headers.get("x-request-id") or str(uuid.uuid4())
    bind_contextvars(
        request_id=request_id,
        method=request.method,
        path=request.url.path,
    )
    response = await call_next(request)
    response.headers["x-request-id"] = request_id
    return response

```

A partir de aquí, un `log.info("stock_reserved")` en cualquier servicio interno lleva el `request_id` sin que esa función sepa nada de HTTP. El `clear_contextvars()` inicial evita que el contexto de una petición anterior se filtre en la siguiente cuando el worker se reutiliza.

Unificar las bibliotecas de terceros: `ProcessorFormatter`

Tu aplicación no es la única que loguea. `uvicorn`, `SQLAlchemy`, `boto3`, `celery`... todas usan el módulo `logging` de la biblioteca estándar, y si no haces nada, sus líneas saldrán en texto plano mezcladas con tu JSON. El resultado es una plataforma de logs medio consultable: tus eventos sí, los de las dependencias no —y justo el `WARNING` del pool de conexiones de `SQLAlchemy` es el que necesitabas agrupar.

`structlog` resuelve esto con `structlog.stdlib.ProcessorFormatter`: un `logging.Formatter` que pasa los registros de la biblioteca estándar por tu misma cadena de procesadores. Todo —tu código con `structlog.get_logger()` y el de terceros con `logging.getLogger()`— acaba saliendo por el mismo pipeline, con el mismo formato JSON y los mismos campos de contexto:

```

import logging
import structlog

shared_processors = [
    structlog.contextvars.merge_contextvars,
    structlog.processors.add_log_level,
    structlog.processors.TimeStamper(fmt="iso", utc=True),
]

```

```

]

structlog.configure(
    processors=shared_processors + [
        # convierte el event_dict en algo que ProcessorFormatter entiende
        structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
    ],
    logger_factory=structlog.stdlib.LoggerFactory(),
    cache_logger_on_first_use=True,
)

formatter = structlog.stdlib.ProcessorFormatter(
    # se aplican solo a los logs que vienen de logging (uvicorn, sqlalchemy...)
    foreign_pre_chain=shared_processors,
    processors=[
        structlog.stdlib.ProcessorFormatter.remove_processors_meta,
        structlog.processors.JSONRenderer(),
    ],
)

handler = logging.StreamHandler()
handler.setFormatter(formatter)
root = logging.getLogger()
root.addHandler(handler)
root.setLevel(logging.INFO)

```

Dos detalles importan aquí. Primero, `wrap_for_formatter` debe ser el último procesador de la configuración de `structlog`: coloca el diccionario del evento en el `LogRecord` para que `ProcessorFormatter` lo renderice una sola vez; sin él acabarías con JSON doblemente codificado. Segundo, `foreign_pre_chain` aplica tus procesadores compartidos (`timestamp`, `nivel`, `contextvars`) también a los logs "extranjeros", así el `request_id` de tu middleware aparece incluso en las líneas de `uvicorn`. Recuerda también silenciar los handlers propios que algunas bibliotecas instalan (`uvicorn` añade los suyos) para no duplicar líneas.

Excepciones con contexto: cómo loggear errores útiles

Un error sin contexto es un misterio; un error con contexto es una tarea. La diferencia entre ambos se decide en el momento de loguearlo. La primera regla: usa el mecanismo de excepciones de tu logger, no `str(e)`. En `structlog`, `log.exception()` (o `exc_info=True`) activa `format_exc_info` y la traza completa viaja serializada en un campo del evento, con la cadena de causas incluida:

```

try:
    charge = provider.charge(order)
except ProviderTimeout as e:
    log.exception(
        "payment_failed",
        order_id=order.id,
        provider="stripe",
        amount_cents=order.amount_cents,
        attempt=attempt,
        timeout_ms=cfg.provider_timeout_ms,
    )
    raise

```

Fíjate en qué campos acompañan al error: los que alguien necesitará para decidir a las tres de la mañana. ¿Qué pedido? ¿Qué proveedor? ¿Qué intento era? ¿Qué timeout estaba configurado? Ese último campo es oro: convierte "falló por timeout" en "falló porque 5.000 ms no bastan cuando el proveedor está degradado", que es una conclusión accionable.

Segunda regla: loguea el error una sola vez, en el nivel que lo gestiona. El antipatrón clásico es capturar, loggear y relanzar en cada capa, generando cuatro trazas idénticas del mismo fallo que cuadruplican el ruido y arruinan cualquier conteo de errores. Si una capa no puede hacer nada con la excepción, que la deje subir sin loguearla; el manejador global del framework la registrará con todo el contexto de la petición.

Tercera: distingue errores esperados de inesperados. Un pago rechazado por fondos insuficientes es un warning con event="payment_declined" —es negocio, no avería—. Una NullPointerException en el mismo flujo es un error con traza. Mezclarlos en el mismo nivel entierra las averías reales bajo el ruido del negocio, y las alertas de "tasa de errores" dejan de significar nada. Si usas un agregador de errores tipo Sentry, esta distinción también decide qué le envías: los inesperados sí, el flujo normal de negocio no.

Node.js: pino, rendimiento y redacción

En Node, pino es la opción por defecto para producción: emite JSON directamente y su overhead es de los más bajos del ecosistema. Configuración base con redacción de secretos incluida:

```
import pino from "pino";

export const logger = pino({
  level: process.env.LOG_LEVEL || "info",
  timestamp: pino.stdTimeFunctions.isoTime,
  base: { service: "checkout-api" },
  redact: {
    paths: [
      "req.headers.authorization",
      "req.headers.cookie",
      "/*.password",
      "/*.token",
    ],
    censor: "[REDACTED]",
  },
});

logger.info({ orderId: "ord_123", amountCents: 4999 }, "order_created");
```

La opción redact merece atención especial: los secretos que se cuelan en los logs acaban indexados en tu plataforma de observabilidad, fuera del alcance de tu gestor de secretos y de sus rotaciones. Redactar en el logger —y no confiar en que cada llamada tenga cuidado— es la única defensa que escala.

Para HTTP, pino-http genera un logger hijo por petición con su request id:

```
import express from "express";
import pinoHttp from "pino-http";
import { randomUUID } from "node:crypto";
import { logger } from "../logger.js";

const app = express();

app.use(
  pinoHttp({
    logger,
    genReqId: (req, res) => {
      const id = req.headers["x-request-id"] || randomUUID();
      res.setHeader("x-request-id", id);
      return id;
    },
  })
);

app.post("/orders", (req, res) => {
  req.log.info({ orderId: "ord_123" }, "order_created");
  res.status(201).end();
});
```

req.log es un child logger: cada línea que emitas con él ya incluye el req.id y los datos de la petición, sin repetirlos a mano.

Rendimiento en serio: transports, worker threads y logs que se pierden

El logging síncrono tiene un coste real: serializar JSON, formatear y escribir en stdout ocurre en el hilo principal, compitiendo con tus peticiones. Pino ataca esto en dos frentes. Primero, la serialización en sí está brutalmente optimizada (esquemas precompilados, nada de inspección dinámica). Segundo, desde la v7 los transports mueven el destino del log a un worker thread: tu aplicación serializa la línea, la pasa por un buffer compartido, y el worker se encarga de enviarla a un fichero, a una API o a OpenTelemetry sin bloquear el event loop.

```
import pino from "pino";

const logger = pino({
  level: "info",
  transport: {
    target: "pino-opentelemetry-transport", // corre en un worker thread
  },
});
```

Esta arquitectura tiene una contrapartida que debes conocer: si el proceso muere de golpe (OOM kill, crash nativo), las líneas que estaban en el buffer del transport pueden perderse —incluidas, con mala suerte, las que explicaban por qué murió. Para los flujos críticos, pino ofrece `pino.destination({ sync: true })` y `transport-level flushSync`; y en cualquier caso conviene capturar señales de terminación y llamar a `logger.flush()` antes de salir.

En Python el equilibrio es distinto: structlog es rápido para lo que hace (la recomendación práctica es activar `cache_logger_on_first_use=True` y usar `make_filtering_bound_logger`, que descarta los niveles desactivados casi gratis), pero la escritura sigue siendo síncrona. Si el volumen aprieta, la respuesta no suele ser un logger asíncrono exótico sino escribir menos: bajar niveles, muestrear y agregar. Y una regla universal en ambos lenguajes: nunca hagas trabajo caro para construir argumentos de un log que puede estar desactivado —nada de `json.dumps(objeto_enorme)` como argumento de un debug.

Correlación con trazas de OpenTelemetry

Si ya usas tracing distribuido, tus logs deberían llevar el `trace_id` y el `span_id` activos. Es el puente entre ambas señales: desde una traza lenta saltas a sus logs exactos, y desde un log de error abres la traza completa que lo produjo. En structlog basta un procesador propio:

```
from opentelemetry import trace

def add_trace_context(logger, method_name, event_dict):
    span = trace.get_current_span()
    ctx = span.get_span_context()
    if ctx.is_valid:
        event_dict["trace_id"] = format(ctx.trace_id, "032x")
        event_dict["span_id"] = format(ctx.span_id, "016x")
    return event_dict

# añádelo a la lista de processors, antes de JSONRenderer
```

En Node ni siquiera necesitas escribirlo: el paquete `@opentelemetry/instrumentation-pino` inyecta `trace_id` y `span_id` en cada línea automáticamente cuando hay un span activo.

OpenTelemetry Logs: exportar por OTLP sin casarte con nadie

Hasta ahora hemos hablado de escribir JSON en stdout y dejar que la plataforma lo recoja. OpenTelemetry propone un paso más: tratar los logs como una señal de primera clase, con su propio modelo de datos (LogRecord con atributos, resource y trace context nativos) y su propio protocolo de exportación (OTLP). La ventaja estratégica es la neutralidad: el mismo pipeline sirve para Loki, Elasticsearch, Datadog o cualquier backend compatible, y

cambiar de proveedor es cambiar la configuración del Collector, no tu código.

En Python, el puente se hace desde el módulo logging estándar con un `LoggingHandler` que traduce cada `LogRecord` al modelo OTel y lo envía en lotes:

```
from opentelemetry._logs import set_logger_provider
from opentelemetry.exporter.otlp.proto.grpc._log_exporter import OTLPLogExporter
from opentelemetry.sdk._logs import LoggerProvider, LoggingHandler
from opentelemetry.sdk._logs.export import BatchLogRecordProcessor
from opentelemetry.sdk.resources import Resource
import logging

provider = LoggerProvider(
    resource=Resource.create({"service.name": "checkout-api"})
)
provider.add_log_record_processor(
    BatchLogRecordProcessor(OTLPLogExporter(endpoint="http://otel-collector:4317"))
)
set_logger_provider(provider)

handler = LoggingHandler(level=logging.INFO, logger_provider=provider)
logging.getLogger().addHandler(handler)
```

Un matiz honesto: la especificación de logs de OTel es estable, pero en el SDK de Python estos módulos siguen bajo el espacio de nombres `_logs`, señal de que la API aún puede cambiar entre versiones menores. Fija versiones y lee los changelogs antes de actualizar.

En Node, `pino-opentelemetry-transport` hace todo el trabajo dentro de un worker thread: arranca un `LoggerProvider` de OTel en el worker, parsea cada línea que emite pino, la traduce al modelo de datos de OTel Logs y la envía por OTLP. Tu hilo principal solo serializa JSON, como siempre. Un patrón de adopción muy razonable es híbrido: seguir escribiendo en `stdout` (tu red de seguridad, siempre legible con `kubectl logs`) y añadir OTLP como segundo destino desde el Collector, no desde la aplicación.

Canonical log lines: un evento ancho por petición

Hay una técnica que multiplica el valor de tus logs sin multiplicar su volumen: la canonical log line, popularizada por Stripe. La idea es que, además de los logs puntuales que emitas durante una petición, cada petición termine con un único evento "ancho" que resume todo lo que pasó: quién llamó, a qué endpoint, con qué resultado, cuánto tardó cada dependencia, qué decisiones de negocio se tomaron.

```
import time
from structlog.contextvars import bind_contextvars

@app.middleware("http")
async def canonical_line(request, call_next):
    start = time.perf_counter()
    response = await call_next(request)
    log.info(
        "http_request",          # el evento canónico, uno por petición
        status=response.status_code,
        duration_ms=round((time.perf_counter() - start) * 1000, 1),
        # los campos de negocio se acumularon en contextvars durante la petición:
        # user_id, plan, cache_hit, db_queries, payment_provider...
    )
    return response

# en cualquier capa de la aplicación, sin tocar firmas:
def apply_discount(order):
    bind_contextvars(discount_code=order.discount, discount_pct=15)
```

La consecuencia práctica: el 90% de las preguntas de un incidente ("¿qué peticiones del plan enterprise fallaron con timeout en el proveedor de pagos?") se responden consultando un solo tipo de evento con muchos campos, en lugar de reconstruir la historia juntando ocho líneas dispersas. Los logs puntuales siguen existiendo para el detalle; la línea canónica es el índice. Si solo puedes permitirte retener una parte de tus logs, retén estas.

Dos consejos de implementación: emite la línea canónica también cuando la petición falla (el middleware debe capturar la excepción, anotar `error_type` y re-lanzarla), y resiste la tentación de meter payloads enteros: campos escalares con nombres estables, no documentos anidados de diez niveles.

Workers y colas: el `request_id` no muere en el broker

Todo lo anterior funciona de maravilla... hasta que la petición encola un trabajo y el contexto se evapora. El consumidor procesa el mensaje media hora después, en otro proceso y otra máquina, y sus logs ya no llevan ni `request_id` ni `trace_id`. La regla es simple: el contexto viaja dentro del mensaje, igual que viajaría en una cabecera HTTP.

```
# Productor: adjunta el contexto al encolar
queue.publish({
  "type": "send_invoice",
  "order_id": "ord_123",
  "_ctx": {"request_id": request_id}, # el contexto viaja con el mensaje
})

# Consumidor (Celery, RQ, consumidor de Kafka...): re-enlaza al procesar
from structlog.contextvars import bind_contextvars, clear_contextvars

def handle(message):
    clear_contextvars()
    bind_contextvars(
        request_id=message.get("_ctx", {}).get("request_id"),
        job_type=message["type"],
        job_id=message["id"],
    )
    log.info("job_started")
    ...
    log.info("job_finished", duration_ms=elapsed)
```

En Node el patrón es idéntico con BullMQ: guarda `request_id` en `job.data` al encolar y crea un child logger en el worker con `logger.child({ requestId: job.data.requestId, jobId: job.id })`. Si usas OpenTelemetry, propaga también el trace context serializado (los propagadores W3C hacen esto por ti en muchas integraciones) para que la traza cruce el broker.

Los eventos `job_started` / `job_finished` / `job_failed` con `job_id`, intento (`attempt`) y duración son la versión asíncrona de la línea canónica: te permiten responder "¿cuántos reintentos hubo esta noche y de qué tipo?" sin salir de la plataforma de logs. Y cuando un trabajo falla tras agotar reintentos, ese log final debe llevar el `request_id` original: es lo que conecta la queja del usuario de esta mañana con el job que murió de madrugada.

Niveles: qué registrar (y qué no)

En producción, el nivel por defecto razonable es `info`. Los eventos de negocio relevantes —registro completado, pedido creado, pago rechazado, despliegue aplicado— van a `info`; las situaciones anómalas pero gestionadas (un reintento, un fallback activado) a `warning`; y error queda reservado para fallos que alguien debería mirar. `debug` se queda apagado salvo investigación activa: el coste de ingestión y el ruido no compensan tenerlo siempre encendido.

Dos reglas ahorran mucho dinero y ruido: no loguees dentro de bucles calientes (agrega y emite un resumen al final), y no uses logs como métricas. Contar peticiones o medir latencias es trabajo de contadores e histogramas; el log aporta el detalle de los casos concretos.

Muestreo y control de costes: loguear menos sin ver menos

La factura de logs crece con el tráfico, no con el valor. Un servicio sano a escala emite millones de líneas idénticas de "todo bien" que pagas por ingerir, indexar y retener. Las técnicas de control, de más simple a más fina:

- Elimina en origen lo que nunca consultarás: health checks (GET /healthz cada 5 segundos × N pods), redirecciones, peticiones de preflight CORS. Un filtro en el shipper o un if en el middleware.
- Muestrea lo repetitivo y exitoso: conservar el 1-10% de los info de rutas calientes mantiene la visibilidad estadística con una fracción del coste. Bien configurado, el muestreo y el throttling pueden recortar el volumen en un 70-90% sin perder capacidad de diagnóstico.
- Nunca muestrees errores ni líneas canónicas: la regla de oro. warning y error pasan siempre; la canonical log line pasa siempre. Lo que se muestrea es el ruido de fondo, no la señal.
- Retención por niveles: los error de 90 días y los info de 7 cuestan una fracción de "todo 90 días". La mayoría de plataformas lo soportan de forma nativa.
- Nivel dinámico sin redespargar: poder subir un servicio concreto a debug durante veinte minutos de investigación (variable de entorno, endpoint de administración o SIGHUP) y volver a bajarlo. Es la alternativa barata a tener debug siempre encendido "por si acaso".

Una heurística útil para decidir qué muestrear: si borraras el 99% de estas líneas, ¿perderías alguna capacidad de respuesta en un incidente? Para "request completed 200 OK en /api/products" la respuesta es no —la latencia y el conteo ya están en tus métricas, y el detalle raro lo cubre la línea canónica.

Debug dirigido: verbosidad quirúrgica en producción

Entre "debug apagado" y "debug para todo el mundo" existe un punto intermedio muy superior a ambos: activar la verbosidad solo para la petición, el usuario o el tenant que estás investigando. El mecanismo es sencillo si ya tienes contexto por petición: una cabecera o un flag (por ejemplo, x-debug-request: 1, emitida solo por herramientas internas, o un feature flag por user_id) que el middleware detecta y usa para bajar el umbral de nivel de esa petición concreta.

```
@app.middleware("http")
async def targeted_debug(request, call_next):
    debug_users = await flags.get("debug_user_ids")    # feature flag
    user_id = get_user_id(request)
    if user_id in debug_users:
        bind_contextvars(debug_session=True)
        # el logger de la petición baja a DEBUG solo aquí
    return await call_next(request)
```

El resultado: el usuario que lleva tres días reportando un bug irreproducible genera logs de nivel debug con todo el detalle, mientras el resto del tráfico sigue en info y la factura ni se entera. El mismo patrón sirve para un canary: los pods de la nueva versión arrancan con más verbosidad durante la primera hora. Dos precauciones: los flags de debug deben caducar solos (un TTL en el flag, no una nota mental), y el nivel debug sigue sin ser excusa para volcar payloads con PII —la redacción del logger aplica igual.

Métricas desde logs: dónde está la frontera

Las plataformas modernas permiten derivar métricas de los logs (log-based metrics en Cloud Logging, recording rules sobre LogQL en Loki, log-to-metrics en Datadog), y la tentación de usar los logs como fuente universal es real. La frontera sana: las métricas nativas (contadores, histogramas instrumentados en el código) son la fuente de verdad para tasas, latencias y saturación —son baratas, precisas y no dependen del pipeline de logs—. Las métricas derivadas de logs brillan para lo que no instrumentaste a tiempo: contar un error_type concreto durante un incidente, medir la frecuencia de un evento de negocio recién descubierto, validar una hipótesis antes de instrumentarla en condiciones.

La trampa a evitar es la dependencia estructural: si tu SLO se calcula desde logs, cualquier cambio en el formato, el muestreo o la retención altera tu SLO sin que nadie toque el servicio. El muestreo del 90% del ruido —perfectamente razonable para diagnóstico— es devastador para una métrica derivada que asumía ver todas las líneas. Regla práctica: las métricas que alimentan alertas y SLOs se instrumentan en el código; las derivadas de logs son herramientas de exploración y puentes temporales, y las líneas canónicas (nunca muestreadas) son la

única base aceptable si no queda más remedio.

Tormentas de logs: cuando el error se repite cien mil veces

Hay un modo de fallo que ningún muestreo estático anticipa: la dependencia caída que convierte cada petición en tres líneas de error y una traza. En minutos, el servicio pasa de 50 líneas por segundo a 20.000, el shipper se atasca, la plataforma aplica rate limiting y —esta es la parte cruel— los logs que descarta el rate limit son exactamente los del incidente que intentas investigar. La tormenta de logs es un incidente dentro del incidente.

Las defensas son conocidas en otros contextos y se aplican igual aquí. Deduplicación con ventana: si el mismo (event, error_type, servicio) se repite, emite la primera aparición completa y después un resumen periódico ("este error ocurrió 4.812 veces más en los últimos 60 s") en lugar de cada instancia —el sampling nativo de zerolog y utilidades equivalentes en otros ecosistemas hacen esto casi gratis. Rate limit por logger: un tope de N errores por segundo por tipo de evento, aplicado en la aplicación o en el shipper (throttle en Fluent Bit, VRL con estado en Vector). Y la versión estructural: si el error viene de una dependencia, el circuit breaker que protege tu latencia también protege tus logs, porque cuando abre deja de generar un fallo (y sus tres líneas) por petición, sustituyéndolos por un único evento circuit_opened.

La prueba de fuego es barata: en staging, tumba una dependencia a propósito y mira cuántas líneas por segundo emite tu servicio. Si la respuesta te parece cara multiplicada por tus pods de producción y por una hora de incidente, ya sabes qué defensa falta. Es el tipo de descubrimiento que prefieres hacer un martes por la tarde y no durante la caída real.

Un esquema común: convenciones semánticas

El logging estructurado solo rinde si los campos significan lo mismo en todos los servicios. Si el equipo A escribe userId, el B user_id y el C uid, cada consulta cruzada necesita tres variantes y las dashboards se llenan de ORs. La solución no es heroica: un documento de una página con el esquema mínimo común, y a ser posible alineado con un estándar existente.

Los dos estándares que importan son las convenciones semánticas de OpenTelemetry (service.name, http.request.method, http.response.status_code, url.path...) y ECS, el Elastic Common Schema —que desde 2023 está convergiendo oficialmente con las convenciones de OTel, así que apostar por OTel es apostar por el punto de encuentro. No necesitas adoptar el catálogo entero: elige la docena de campos que cruzan servicios (identificadores, HTTP, errores) y deja libertad en el resto.

```
// Esquema mínimo acordado (ejemplo)
{
  "timestamp": "...",           // ISO 8601 UTC, siempre
  "level": "info",              // debug | info | warning | error
  "event": "order_created",     // snake_case, catálogo documentado
  "service": "checkout-api",    // nombre canónico del servicio
  "version": "2026.07.15",     // versión desplegada
  "request_id": "...",         // propagado desde el borde
  "trace_id": "...",          // si hay tracing
  "user_id": 42,               // numérico, no "user-42"
  "error.type": "TimeoutError", // solo en errores
  "error.message": "...",      // solo en errores
}
```

Dos detalles que ahorran disgustos: fija también los tipos (un user_id que unas veces es número y otras cadena rompe los índices tipados de Elasticsearch), y versiona el esquema en un repositorio con revisión de cambios, porque renombrar un campo consultado por veinte dashboards es un incidente en sí mismo.

Kubernetes por dentro: qué pasa entre tu stdout y el shipper

Escribir en stdout es la mitad de la historia; conviene saber qué ocurre después, porque varios "misterios" de

producción viven ahí. El runtime de contenedores (containerd, CRI-O) captura el stdout de cada contenedor y lo escribe en un fichero del nodo (/var/log/pods/...) en formato CRI: cada línea tuya se envuelve con timestamp, stream y una marca de si la línea fue partida. Y aquí llega el primer matiz: el runtime parte las líneas que superan su límite (16 KB en containerd por defecto), así que un evento JSON gigante puede llegar troceado. Los shippers saben recomponerlo (la opción de parseo CRI en Fluent Bit), pero es otra razón para no loguear payloads enormes.

Segundo matiz: la rotación. El kubelet rota estos ficheros por tamaño (containerLogMaxSize, 10 MB por defecto) y conserva unas pocas rotaciones. Si tu servicio emite ráfagas masivas —un error en bucle, por ejemplo— la rotación puede adelantar al shipper y esas líneas se pierden sin dejar rastro. La defensa es doble: no loguear en bucle (un circuit breaker de logging, o al menos un contador agregado) y dimensionar el buffer del shipper con margen.

Tercero: kubectl logs lee esos mismos ficheros del nodo, no tu plataforma. Por eso funciona aunque el pipeline de observabilidad esté caído —es tu red de seguridad— pero también por eso solo muestra lo que aún no se ha rotado, y nada de pods ya eliminados. Para la historia completa, la plataforma; para el vistazo inmediato al pod que acaba de arrancar mal, kubectl logs --previous es a menudo la herramienta más rápida que existe.

Del stdout a tu plataforma: Fluent Bit, Vector y el Collector

Tu aplicación escribe JSON en stdout y ahí acaba su responsabilidad. El viaje hasta la plataforma de logs lo hace un shipper, y en Kubernetes el patrón dominante es un DaemonSet que lee los ficheros de contenedor de cada nodo, enriquece cada línea con metadatos (pod, namespace, labels) y la envía al backend. Las tres opciones serias:

- Fluent Bit: el estándar ligero (20-30 MB de memoria por nodo, grafo CNCF). Parsea, filtra, enriquece con el filtro de Kubernetes y envía a prácticamente cualquier backend. Para la mayoría de clusters es la elección por defecto.
- Vector: escrito en Rust, más capaz en transformaciones (su lenguaje VRL permite remodelar, redactar y muestrear con lógica real) y con semántica de entrega garantizada con acknowledgements de extremo a extremo. Consume más (50-80 MB típicos) y tiene una arista conocida: si un pod se borra mientras Vector aún tiene backlog de sus logs, los metadatos de Kubernetes de ese pod pueden perderse.
- OpenTelemetry Collector: la opción natural si ya lo usas para trazas y métricas —un solo agente para las tres señales, con el receptor filelog para los logs de contenedor.

```
# Fluent Bit: parsear JSON de contenedores y muestrear ruido (extracto)
[FILTER]
    Name    kubernetes
    Match   kube.*
    Merge_Log On          # el JSON de la app se convierte en campos de primer nivel

[FILTER]
    Name    grep
    Match   kube.*
    Exclude $path /healthz # los health checks no viajan

[OUTPUT]
    Name    loki
    Match   kube.*
    labels  job=fluentbit, $kubernetes['namespace_name']
```

Dos consejos con cualquier shipper. Primero, cuidado con las etiquetas/índices de alta cardinalidad: en Loki, cada combinación de labels crea un stream —etiqueta por namespace y servicio, nunca por request_id o user_id (esos se buscan con filtros, no con labels). Segundo, resuelve las stack traces multilinea en el borde si alguna aplicación aún no emite JSON: los parsers multiline de Fluent Bit existen precisamente porque una traza de Java partida en 40 logs sueltos es inservible.

Cuando el pipeline se cae: buffering, backpressure y entrega

El pipeline de logs también falla: la plataforma tiene un incidente, la red se congestiona, el backend aplica rate limiting. Lo que ocurra entonces depende de decisiones que conviene tomar en frío. La primera es el buffer del shipper: en memoria (rápido, se pierde si el agente muere) o en disco (sobrevive reinicios, consume I/O del nodo). Para logs de aplicación, un buffer híbrido con desborde a disco y un límite claro es el equilibrio habitual; para audit logs, disco siempre. La segunda es la política de desborde cuando el buffer se llena: ¿descartar lo más viejo, descartar lo nuevo, o frenar? Fluent Bit y Vector permiten elegir, y la respuesta correcta depende del flujo —descartar debug antiguo es aceptable; descartar errores recientes, no—.

La tercera decisión es qué garantía pides de extremo a extremo. Vector ofrece acknowledgements end-to-end: el agente no da por entregada una línea hasta que el backend la confirma, lo que convierte "creo que llegó" en "llegó". El precio es más memoria y la posibilidad de backpressure: si el backend va lento, la cola crece hacia atrás. Lo importante es que el backpressure nunca alcance a tu aplicación —escribir a stdout no debe bloquearse jamás por culpa del pipeline; ese aislamiento es precisamente la razón de arquitectura por la que el shipper es un proceso separado y no una biblioteca dentro de tu app.

Y cierra el círculo la observabilidad del propio pipeline: el shipper exporta métricas (líneas leídas, enviadas, descartadas, tamaño del buffer, reintentos) y merecen una alerta propia. Un buffer que crece de forma sostenida avisa de un backend degradado horas antes de que se pierda la primera línea; un contador de descartes distinto de cero durante un incidente explica por qué "faltan logs" en la investigación posterior. La pregunta "¿podemos confiar en que los logs de anoche están completos?" debe tener respuesta comprobable, no esperanzada.

Serverless: logging sin daemon que te salve

En AWS Lambda, Cloud Run o Azure Functions no hay DaemonSet ni sidecar: el stdout va directo al servicio de logs de la plataforma (CloudWatch Logs, Cloud Logging), y tu única palanca es lo que escribas. Las reglas cambian un poco. Primera: el JSON estructurado importa aún más, porque CloudWatch Logs Insights y Cloud Logging descubren los campos JSON automáticamente y te dan consultas tipadas gratis; Lambda incluso soporta formato de log JSON nativo, configurable por función, que estructura también los logs del propio runtime.

Segunda: el contexto de la invocación es tu request_id. En Lambda, el context.aws_request_id (y el trace id de X-Ray o de OTel si lo usas) debe ir enlazado en cada línea igual que harías en un middleware HTTP —mismo patrón de contextvars, distinto punto de entrada. Tercera: cuidado con el flush. El entorno se congela en cuanto el handler devuelve; los transports asíncronos con buffer (incluido el de OTLP) pueden perder las últimas líneas si no fuerzas el vaciado antes de retornar. En serverless, la escritura síncrona a stdout no es una limitación: es la opción correcta.

Y una advertencia económica: CloudWatch cobra la ingesta a un precio que convierte el debug olvidado en la función caliente en una partida visible de la factura mensual. Las técnicas de la sección de costes —nivel por defecto info, muestreo del ruido, retención corta para lo verboso— aplican aquí con más motivo, y la retención por grupo de logs es de un clic.

¿Y fuera de Python y Node?

Los patrones de esta guía son agnósticos del lenguaje; solo cambian las bibliotecas. En Go, el propio stdlib incluye desde la 1.21 el paquete log/slog con salida JSON, atributos tipados y handlers componibles —para servicios nuevos ya no hay excusa para el log.Printf—, y zerolog sigue siendo la referencia cuando el rendimiento es crítico. En Java, la combinación habitual es SLF4J/Logback con logstash-logback-encoder para JSON y MDC para el contexto por petición (el equivalente de los contextvars). En .NET, Serilog con enrichers cumple el mismo papel, y Microsoft.Extensions.Logging soporta scopes estructurados de serie. En Rust, tracing unifica logs y spans en un solo modelo, lo que encaja de fábrica con la filosofía de correlación de esta guía.

La consecuencia práctica para equipos políglotas: el documento de esquema común (campos, tipos, nombres de

evento) pesa más que cualquier elección de biblioteca. Si el servicio de Go llama al campo `user_id` y el de Java `userId`, habrás reproducido el problema que el logging estructurado venía a resolver, con JSON perfectamente válido en ambos lados.

Multi-tenant: el `tenant_id` es un campo de primera clase

Si tu producto sirve a varios clientes desde la misma infraestructura, el `tenant_id` merece el mismo trato que el `request_id`: se enlaza en el borde (desde el token o el subdominio), viaja por `contextvars` y aparece en cada línea, incluidas las de los workers. Los beneficios son inmediatos: "¿este incidente afecta a todos los clientes o solo a uno?" es la primera pregunta de cualquier guardia en un SaaS, y con `tenant_id` en la línea canónica se responde con un `group by`. También habilita el análisis de coste por cliente (¿qué `tenant` genera el 40% del volumen de logs?) y el soporte dirigido: el debug quirúrgico de la sección anterior aplicado a un `tenant` concreto durante una investigación.

Las precauciones son dos. En plataformas con índices por etiqueta como Loki, `tenant_id` como label solo es viable con pocos tenants (decenas); con miles, es un campo filtrable, no un label —la cardinalidad de nuevo—. Y si tus contratos incluyen aislamiento de datos o residencia, los logs cuentan como datos del cliente: el enrutamiento por `tenant` en el shipper (streams o buckets separados) es el punto natural donde aplicarlo, otra razón para que el `tenant_id` sea un campo fiable y no una convención a medias.

Seguridad y cumplimiento: más allá de redactar

La redacción en el logger es la primera línea, no la última. Un programa completo tiene cuatro capas. Primera: clasifica qué es PII en tu dominio (emails, IPs, nombres, direcciones, tokens, y combinaciones que identifican indirectamente) y decide campo a campo: ¿se loguea, se redacta, se transforma? Segunda: transforma en origen cuando necesites correlacionar sin exponer —un HMAC con clave del email (`user_hash`) permite agrupar todos los logs del mismo usuario sin almacenar el email; truncar la IP (192.168.1.0/24) suele bastar para el análisis geográfico. Tercera: una red de seguridad en el shipper (VRL en Vector, filtros lua en Fluent Bit, o el procesador de transformación del Collector) que capture patrones de tarjetas, emails o tokens que se colaron pese a todo. Cuarta: retención y acceso —los logs con datos personales tienen las mismas obligaciones GDPR que tu base de datos, pero sin sus herramientas de borrado selectivo, así que la retención corta es tu mecanismo de cumplimiento más realista.

Mención aparte merecen los audit logs (quién accedió a qué, cambios de permisos, acciones administrativas): esos tienen requisitos opuestos a los logs de aplicación —retención larga, inmutabilidad, acceso restringido— y conviene separarlos en un flujo propio desde el día uno, no pescarlos después entre el ruido operacional.

Los logs también se revisan: el evento como contrato de equipo

Un catálogo de eventos vivo es lo que separa "logging estructurado" de "JSON estructurado con caos dentro". Funciona igual que el resto del contrato del servicio: los nombres de evento y sus campos viven en un documento versionado (una página en el repositorio basta), y añadir o cambiar un evento pasa por revisión de PR como cualquier cambio de API. La revisión no necesita ser burocrática; tres preguntas bastan. ¿El nombre sigue la convención (`snake_case`, verbo en pasado, sin identificadores dentro)? ¿Los campos nuevos reutilizan nombres y tipos del esquema común? ¿Este log aporta algo que las métricas y los eventos existentes no cubren ya?

Esa tercera pregunta es la más valiosa, porque el impulso natural de todo equipo es loggear más "por si acaso", y cada línea nueva es coste perpetuo de ingestión y ruido. La disciplina inversa también aplica: cuando un evento deja de consultarse (las plataformas suelen mostrar estadísticas de uso de campos y saved queries), se deprecia y se elimina. Un catálogo de cuarenta eventos bien nombrados que todo el mundo conoce vale más que cuatrocientos que nadie recuerda. Y un beneficio lateral que se nota en el onboarding: el catálogo de eventos es, en la práctica, un mapa de qué hace el sistema —leerlo es una de las formas más rápidas de entender un servicio nuevo.

Complemento natural del catálogo: loguea la configuración al arrancar. Un único evento `service_started` con la versión desplegada, los flags activos y los valores de configuración relevantes (redactados donde toque) convierte la pregunta "¿qué configuración tenía el pod cuando pasó esto?" en una búsqueda de un segundo, y es la primera pista cuando un despliegue cambia el comportamiento sin cambiar el código.

Logging en aplicaciones con LLM: tokens, coste y privacidad

Si tu servicio llama a un modelo de lenguaje, tus logs tienen un trabajo nuevo: explicar qué pidió el usuario, con qué modelo se respondió, cuánto costó y por qué falló. Una llamada a un LLM es una dependencia externa cara, lenta y no determinista: exactamente el tipo de operación que más justifica un evento estructurado por invocación. Y sin embargo, la mayoría de los equipos que ya loguean bien sus peticiones HTTP tratan las llamadas al modelo como una caja negra de la que solo queda un INFO Calling OpenAI... en texto libre.

OpenTelemetry ya define convenciones semánticas para IA generativa: los atributos `gen_ai.*` cubren proveedor, modelo solicitado y modelo real de la respuesta, tokens de entrada y salida, y motivo de finalización. Siguen marcadas como experimentales —el trabajo está a medio camino y en 2026 se movieron a su propio repositorio—, pero los principales vendors de observabilidad ya las soportan. Adoptar sus nombres hoy (`gen_ai.request.model`, `gen_ai.usage.input_tokens`, `gen_ai.usage.output_tokens`) te ahorra renombrar campos mañana, aunque no uses OTel para nada más.

El evento mínimo por llamada debería responder cuatro preguntas: qué modelo y con qué parámetros; cuántos tokens y a qué coste; cuánto tardó; y cómo terminó (parada natural, límite de tokens, filtro de contenido o error). Así se ve con `structlog`:

```
import hashlib
import time

import structlog

log = structlog.get_logger()

# Precio por token del modelo (mantenlo en configuración, no hardcodeado)
IN_PRICE = 3.0 / 1_000_000 # USD por token de entrada
OUT_PRICE = 15.0 / 1_000_000 # USD por token de salida

async def call_llm(client, *, model, messages, **params):
    prompt_text = "".join(m["content"] for m in messages)
    prompt_hash = hashlib.sha256(prompt_text.encode()).hexdigest()[:16]
    start = time.monotonic()
    try:
        resp = await client.chat.completions.create(
            model=model, messages=messages, **params
        )
    except Exception as exc:
        log.error(
            "llm_call_failed",
            model=model,
            prompt_hash=prompt_hash,
            prompt_chars=len(prompt_text),
            duration_ms=int((time.monotonic() - start) * 1000),
            error_type=type(exc).__name__,
        )
        raise

    usage = resp.usage
    log.info(
        "llm_call",
        **{
            "gen_ai.request.model": model,
            "gen_ai.response.model": resp.model,
            "gen_ai.usage.input_tokens": usage.prompt_tokens,
            "gen_ai.usage.output_tokens": usage.completion_tokens,
        },
        finish_reason=resp.choices[0].finish_reason,
        prompt_hash=prompt_hash,
```

```

    prompt_chars=len(prompt_text),
    duration_ms=int((time.monotonic() - start) * 1000),
    cost_usd=round(
        usage.prompt_tokens * IN_PRICE
        + usage.completion_tokens * OUT_PRICE, 6
    ),
)
return resp

```

Fíjate en lo que el evento no incluye: ni el prompt ni la respuesta en crudo. Los prompts contienen datos de usuario con una frecuencia altísima —nombres, correos, historiales médicos, contratos— y un log general con retención de 30 días es el peor sitio posible para guardarlos. El hash y la longitud bastan para agrupar llamadas idénticas, detectar prompts anómalamente largos y correlacionar reintentos. Si de verdad necesitas inspeccionar payloads (para depurar calidad o construir evals), captúralos muestreados y con redacción en un almacén aparte con retención corta y acceso restringido, no en el pipeline de logs compartido.

El `finish_reason` merece una mención: es el campo que más incidentes de calidad explica y el que casi nadie loguea. Un pico de `length` significa respuestas truncadas por límite de tokens; un pico de `content_filter`, que el proveedor está bloqueando salidas. Ambos son invisibles si solo mides latencia y errores HTTP.

Para agentes, el patrón de canonical log line que vimos antes encaja de maravilla: además del evento por llamada, emite un evento ancho por ejecución completa del agente, con los totales acumulados:

```

{
  "event": "agent_run_completed",
  "run_id": "run_9f2c",
  "request_id": "3f2a91c0-7b2e-4d1a-9c55-8f0f2b6a1e77",
  "gen_ai.request.model": "claude-sonnet-5",
  "steps": 4,
  "tool_calls": ["search_db", "send_email"],
  "input_tokens_total": 8210,
  "output_tokens_total": 1467,
  "cost_usd": 0.041,
  "duration_ms": 9302,
  "outcome": "success"
}

```

Con ese evento, preguntas como «cuánto nos cuesta al día el agente de soporte», «qué porcentaje de ejecuciones agota el máximo de pasos» o «qué herramienta falla más» son una consulta de agregación, no una investigación. Y cuando el producto pregunte por qué la factura del proveedor subió un 40% este mes, la respuesta estará en un `sum by (gen_ai.request.model)` en lugar de en una hoja de cálculo reconstruida a mano.

Del navegador al backend: la correlación no empieza en tu API

Un usuario reporta «no funciona», abres los logs del backend y todo está en verde. El fallo ocurrió en el navegador: un fetch que el adblocker bloqueó, una excepción de JavaScript, un timeout que el cliente no reintentó. Si tu observabilidad empieza en el balanceador, la primera mitad de cada petición es territorio desconocido.

Cerrar ese hueco tiene dos piezas. La primera es propagar el contexto de traza desde el navegador: si el fetch del frontend envía la cabecera `traceparent` del estándar W3C Trace Context, el `trace_id` que tu backend ya inyecta en cada log (como vimos en la sección de OpenTelemetry) conecta la acción del usuario con todos los servicios que tocó. Las auto-instrumentaciones de OpenTelemetry para JavaScript hacen esto solas; si no quieres cargar el SDK completo, generarla a mano son diez líneas:

```

// Propagar traceparent (W3C Trace Context) en cada fetch
function traceparent() {
  const hex = (n) =>
    Array.from(crypto.getRandomValues(new Uint8Array(n)))
      .map((b) => b.toString(16).padStart(2, "0"))
      .join("");
  return "00-" + hex(16) + "-" + hex(8) + "-01";
}

```

```

}

const _fetch = window.fetch;
window.fetch = (url, opts = {}) => {
  opts.headers = { ...(opts.headers || {}), traceparent: traceparent() };
  return _fetch(url, opts);
};

```

La segunda pieza es enviar los eventos del navegador como logs estructurados. Un logger de cliente decente acumula eventos en memoria y los envía por lotes con `navigator.sendBeacon`, que sobrevive al cierre de la pestaña:

```

const buffer = [];

export function clientLog(event, fields = {}) {
  buffer.push({
    event,
    ts: Date.now(),
    url: location.pathname,
    session_id: sessionId,
    ...fields,
  });
  if (buffer.length >= 20) flush();
}

function flush() {
  if (!buffer.length) return;
  navigator.sendBeacon("/client-logs", JSON.stringify(buffer.splice(0)));
}

addEventListener("visibilitychange", () => {
  if (document.visibilityState === "hidden") flush();
});

window.addEventListener("error", (e) =>
  clientLog("js_error", {
    message: e.message,
    source: e.filename,
    line: e.lineno,
  })
);

```

En el backend, el endpoint `/client-logs` valida el lote, lo enriquece con lo que el navegador no sabe o puede falsear (IP truncada, user agent parseado, user_id de la sesión autenticada) y lo emite por el mismo pipeline que el resto de tus logs, con service: "frontend". Si prefieres no construirlo, SDKs como Grafana Faro empaquetan exactamente este patrón: captura de errores, Web Vitals y logs con correlación de trazas incluida.

Los errores típicos de esta integración son conocidos y evitables. Sirve el endpoint desde tu propio dominio y con un nombre neutro —las rutas tipo `/collect` o `/analytics` mueren a manos de los bloqueadores—. Muestrea agresivamente: mil pestañas abiertas emitiendo eventos de scroll pueden generar más volumen que tu API entera. Trunca los stack traces antes de enviarlos. Y nunca loguees la URL completa con sus query params: los tokens de invitación, los códigos de OAuth y los identificadores personales viven ahí con una frecuencia alarmante.

Alertas sobre logs: del panel bonito al aviso a las 3:07

Un log consultable que nadie consulta hasta que un cliente se queja es potencial desperdiciado. La última pieza del pipeline es alertar sobre los propios logs, y es donde el trabajo de estructurar cada evento se paga solo: alertar sobre `level="error"` a secas es imposible de afinar, pero alertar sobre la tasa de `payment_failed` respecto al total de intentos de pago es una señal de negocio.

En el ecosistema de Grafana, el Ruler de Loki evalúa consultas LogQL de forma periódica y envía las alertas resultantes a Alertmanager, exactamente igual que Prometheus con las métricas. Una regla razonable para el caso del pago:

```
groups:
- name: checkout-alerts
  rules:
  - alert: PaymentErrorRateHigh
    expr: |
      sum(rate({app="checkout-api"} | json
        | level = "error" | event =~ "payment_.*" [5m]))
      /
      sum(rate({app="checkout-api"} | json
        | event =~ "payment_.*" [5m]))
      > 0.05
    for: 10m
    labels:
      severity: page
    annotations:
      summary: "Más del 5% de eventos de pago fallando"
      runbook: "https://wiki.internal/runbooks/payments"
```

Los principios que hacen que una alerta sobre logs funcione son los mismos que con métricas, con un par de matices propios. Alerta sobre proporciones, no sobre conteos absolutos: cinco errores por minuto son gravísimos con diez peticiones y ruido con diez mil. Usa `for` para exigir que la condición persista y no despertar a nadie por un blip de treinta segundos. Y alerta sobre síntomas que un usuario notaría (pagos fallando, latencia p99, colas creciendo), no sobre causas internas que quizá no tengan impacto: las causas van al dashboard de diagnóstico, no al buscapersonas.

El matiz específico de los logs es la cardinalidad: la alerta por tenant, por cliente o por combinación rara de campos —carísima como métrica de Prometheus porque explota el número de series— sale casi gratis como consulta sobre logs. Si una regla LogQL se vuelve demasiado costosa de evaluar cada minuto, conviértela en recording rule: el Ruler la materializa como métrica vía remote write y la alerta pasa a evaluarse sobre Prometheus, que para eso es barato.

Cuánto cuesta un log: presupuesto antes que sorpresa

Terminemos con la conversación incómoda. El logging estructurado multiplica el valor de cada evento, pero también hace muy fácil emitir muchos más, y la factura de observabilidad tiene fama de ser la segunda sorpresa del año después de la del proveedor de LLM. La defensa es aritmética hecha antes, no después:

```
200 req/s × 3 eventos/req × 600 bytes ~ 360 KB/s
360 KB/s × 86.400 s ~ 31 GB/día
31 GB/día × 30 días de retención ~ 930 GB almacenados
930 GB × (precio ingesta + retención) = tu factura mensual
```

Con esos números delante, las palancas de las secciones anteriores dejan de ser teoría: el muestreo de logs de éxito reduce el primer factor; las canonical log lines reducen el segundo (un evento ancho en vez de ocho estrechos); no loguear payloads reduce el tercero. La retención escalonada ataca el cuarto: pocos días en almacenamiento caliente consultable al instante, unas semanas en frío barato tipo objeto, y archivado solo lo que el cumplimiento exija. La mayoría de las consultas de un incidente miran las últimas horas; pagar almacenamiento premium por logs de hace noventa días es pagar un Ferrari para ir al buzón.

Dos prácticas de equipo cierran el círculo. Primera: presupuesto de volumen por servicio, visible en un dashboard, con el mismo estatus que el presupuesto de latencia; un despliegue que triplica los eventos emitidos debería cantar en la revisión igual que uno que triplica el p99. Segunda: el filtrado en la puerta de entrada —Vector o Fluent Bit descartando eventos de debug olvidados, health checks y bots antes de que lleguen a ingesta— es la optimización con mejor ratio esfuerzo/ahorro de todo el pipeline. El objetivo nunca es loguear menos por loguear menos: es que cada gigabyte que pagas responda preguntas que alguien hace.

Evolucionar el esquema sin romper los dashboards

El catálogo de eventos de la sección anterior resuelve el presente; esta resuelve el futuro. Todo esquema de logs cambia —campos que se renombran, tipos que se corrigen, eventos que se dividen— y cada cambio tiene consumidores silenciosos: dashboards, alertas del Ruler, consultas guardadas y ese script de finanzas que nadie recuerda. Romperlos no da error en ningún despliegue; simplemente un panel se queda a cero y una alerta deja de dispararse, que es la peor forma de fallar.

Las reglas que evitan ese escenario son las mismas que en cualquier API pública. Los cambios aditivos —añadir un campo nuevo— son siempre seguros y no requieren ceremonia. Los destructivos —renombrar user a user_id, cambiar amount de string a número, partir checkout_done en dos eventos— exigen una ventana de doble emisión: el servicio emite el campo viejo y el nuevo a la vez, migras los consumidores con calma, y solo entonces retiras el viejo. Dos semanas de campo duplicado son infinitamente más baratas que una alerta de pagos ciega durante un incidente.

Para los eventos críticos —los que alimentan alertas que despiertan a alguien o informes que salen del equipo— merece la pena fijar el contrato en CI, igual que se hace con los esquemas de una API. Un test que valida los eventos emitidos contra el catálogo convierte «alguien renombró un campo sin avisar» de incidente en fallo de build:

```
import jsonschema

PAYMENT_FAILED_V1 = {
    "type": "object",
    "required": [
        "event", "request_id", "order_id",
        "amount_cents", "error_code",
    ],
    "properties": {
        "event": {"const": "payment_failed"},
        "amount_cents": {"type": "integer"},
        "error_code": {"type": "string"},
    },
}

def test_payment_failed_matches_contract(capture_logs):
    trigger_failed_payment()
    event = capture_logs.find(event="payment_failed")
    jsonschema.validate(event, PAYMENT_FAILED_V1)
```

Si un evento cambia de forma incompatible y la doble emisión no es viable, el campo event_version es el plan B honesto: payment_failed con event_version: 2 permite que las consultas distingan épocas y que los dashboards muestren la costura en vez de esconderla. Lo que no funciona es la mutación silenciosa: el mismo nombre de evento con dos formas distintas según la fecha del despliegue es una trampa para el compañero que investigue un incidente dentro de seis meses.

La versión barata de todo esto, si tu equipo es pequeño: antes de fusionar un cambio que toca campos de un evento existente, busca su nombre en las consultas guardadas y las reglas de alerta. Cinco minutos de grep en el repositorio de dashboards evitan la mayoría de las roturas, sin infraestructura nueva.

Cómo testear tus logs

Los logs son una interfaz más de tu sistema —las alertas y los runbooks dependen de ellos— y merecen tests como cualquier interfaz. No hace falta testear cada línea: sí los eventos de los que dependen alertas, dashboards o auditorías. structlog lo pone fácil con capture_logs:

```
from structlog.testing import capture_logs

def test_failed_payment_emits_event():
    with capture_logs() as logs:
        process_payment(order_with_bad_card)

    events = [l for l in logs if l["event"] == "payment_failed"]
```

```

assert len(events) == 1
assert events[0]["order_id"] == "ord_123"
assert events[0]["log_level"] == "warning"
assert "card_number" not in events[0] # la redacción también se testea

```

En Node, el patrón equivalente es inyectar un stream de test en pino y aseverar sobre los objetos parseados:

```

import pino from "pino";

test("failed payment emits event", () => {
  const lines = [];
  const logger = pino({ level: "info" }, {
    write(chunk) { lines.push(JSON.parse(chunk)); },
  });

  processPayment(orderWithBadCard, logger);

  const evt = lines.find((l) => l.msg === "payment_failed");
  expect(evt.orderId).toBe("ord_123");
  expect(evt.cardNumber).toBeUndefined(); // redactado
});

```

El test más valioso de todos es el que verifica que la redacción funciona: es barato, corre en cada CI, y la alternativa es descubrir en una auditoría que llevabas seis meses indexando números de tarjeta. Añade también un test de humo del esquema (todos los eventos llevan timestamp, level, event, service) y habrás blindado el contrato que tus dashboards asumen.

Errores comunes que se pagan caros

- Doble serialización: pasar un JSON ya serializado como mensaje produce JSON dentro de JSON, imposible de consultar. Pasa siempre objetos y campos, nunca cadenas preformateadas.
- Cardinalidad sin control en nombres de evento: "user_42_order_failed" crea un evento nuevo por usuario. El identificador va en un campo; el nombre del evento es fijo.
- PII en los logs: emails, tokens, números de tarjeta. Además del riesgo de seguridad, complica el cumplimiento (GDPR): un log indexado no se puede "borrar selectivamente" con facilidad. Redacta en origen.
- Stack traces multilínea en texto plano: cada línea llega a la plataforma como un log suelto. Con JSON, la excepción completa viaja en un campo del mismo evento.
- Formatos distintos por servicio: si cada equipo nombra los campos a su manera (userId, user_id, uid), las consultas cruzadas mueren. Acordad un esquema mínimo común y documentadlo.
- Confiar en que "ya lo quitaremos": los console.log y print de depuración siempre llegan a producción. Un linter que los prohíba (no-console en ESLint, flake8-print) cuesta cinco minutos y lo evita para siempre.
- Tipos inconsistentes en el mismo campo: un duration_ms que unas veces es 123 y otras "123ms" rompe agregaciones y, en backends con índices tipados, descarta documentos enteros. Fija los tipos en el esquema.
- Logs como única señal: si tu única forma de saber que el servicio va mal es buscar errores en los logs, descubres los incidentes tarde. Las métricas alertan; los logs explican; las trazas localizan.

Consultas que conviene dominar

El logging estructurado se amortiza en el lenguaje de consulta. Merece la pena practicar tres o cuatro patrones antes del incidente, no durante. En Loki (LogQL), el pipeline típico es seleccionar por labels, parsear JSON y filtrar por campos:

```
# errores agrupados por tipo en los últimos 30 min
sum by (error_type) (
  count_over_time(
    {service="checkout-api"} | json | level="error" [30m]
  )
)

# latencia p95 desde las líneas canónicas
quantile_over_time(0.95,
  {service="checkout-api"} | json | event="http_request"
  | unwrap duration_ms [5m]
)
```

En CloudWatch Logs Insights, el mismo espíritu con otra sintaxis:

```
fields @timestamp, request_id, error_type
| filter level = "error" and service = "checkout-api"
| stats count(*) as errores by error_type, bin(5m)
| sort errores desc
```

Y en Elasticsearch/OpenSearch, los campos JSON indexados permiten agregaciones directas (terms sobre `error_type`, percentiles sobre `duration_ms`) desde Kibana sin escribir DSL a mano. Tres observaciones transversales. Una: todas estas consultas dependen de que `level`, `event` y los campos numéricos existan y tengan tipos consistentes —el esquema común es lo que las hace posibles. Dos: las consultas que uses en incidentes deben estar guardadas (saved queries, dashboards) porque a las 3 de la mañana nadie recuerda la sintaxis de `unwrap`. Tres: si una consulta de logs se ejecuta cada cinco minutos para alimentar una alerta, probablemente debería ser una métrica; las alertas sobre logs son legítimas para patrones concretos (un `error_type` específico, un evento de auditoría), no como sustituto barato del sistema de métricas.

Migrar un servicio legacy sin reescribirlo

Pocas veces se parte de cero. El camino incremental que funciona: primero, cambia el formateador a JSON sin tocar ninguna llamada —en Python, `ProcessorFormatter` hace exactamente esto: los `logging.getLogger()` existentes empiezan a salir como JSON con timestamp y nivel correctos sin modificar una línea de código de aplicación. Segundo, añade el middleware de contexto: desde ese momento, hasta los logs legacy llevan `request_id`, que es el 80% del valor. Tercero, ve convirtiendo los mensajes de más valor a eventos con campos ("`Order %s created for %s`" pasa a `event="order_created"` con `order_id` y `user_id`), empezando por los que aparecen en tus búsquedas de incidentes recientes —el historial de búsquedas de tu plataforma es literalmente la lista priorizada de qué migrar. Cuarto, añade la línea canónica, que no requiere tocar código existente, solo el middleware.

Lo que no funciona: la migración big-bang que renombra todos los mensajes a la vez (rompe los dashboards y saved queries que dependían de los textos antiguos, todos el mismo día) y el "nuevo estándar" que solo aplica a servicios nuevos (garantiza dos formatos para siempre). Trata los nombres de evento como una API con deprecación: si `order_created` debe pasar a `checkout_order_created`, emite ambos durante un sprint y avisa a quien tenga dashboards encima.

Caso práctico: reconstruir un fallo de pago a las 3:07

Veamos todo lo anterior trabajando junto. A las 3:07 salta la alerta: la tasa de errores 5xx de `checkout-api` supera el umbral. Abres la plataforma de logs y consultas las líneas canónicas fallidas de los últimos 15 minutos. En Loki:

```
{service="checkout-api"} | json
| event = "http_request" | status >= 500
| line_format "{{.request_id}} {{.path}} {{.error_type}}"
```

Resultado: 47 peticiones fallidas, todas en `POST /orders`, todas con `error_type=TimeoutError` y `payment_provider=stripe`. Ya sabes el qué; falta el dónde. Tomas un `request_id` concreto y pides su historia

completa, a través de todos los servicios:

```
{namespace="prod"} | json  
| request_id = "3f2a91c0-7b2e-4d1a-9c55-8f0f2b6a1e77"
```

Aparecen, ordenadas por timestamp, las líneas de checkout-api (order_created, payment_started), la de payments-svc (provider_request_sent... y nada más durante 10 segundos), y el error final del timeout. La misma línea trae trace_id: un clic y estás viendo la traza en tu backend de tracing, donde el span del proveedor externo muestra los 10.000 ms exactos. La causa: el proveedor de pagos degradado. La confirmación llega de los eventos de warning del circuit breaker de payments-svc —que aparecen en la misma búsqueda porque todos los servicios comparten esquema.

Tiempo total: unos minutos, tres consultas, cero grep. Cada pieza de esta guía jugó su papel: la línea canónica dio el patrón (agrupar por error_type y payment_provider), el request_id propagado unió los servicios, el trace_id saltó a la traza, el esquema común permitió una sola consulta cross-service, y los timeouts logueados con campos numéricos permitieron ver la distribución. Sin logging estructurado, esta misma investigación son dos horas de grep sobre cuatro formatos distintos.

El eco del incidente también pasa por los logs. Para el postmortem, la secuencia completa de eventos con timestamps exactos ya está escrita —la cronología se copia de la consulta, no se reconstruye de memoria—. Y las acciones de seguimiento salen del mismo material: el timeout de 10 s del proveedor era demasiado generoso (el campo timeout_ms lo delató), faltaba una alerta sobre los warning del circuit breaker, y el evento provider_request_sent merecía un campo provider_latency_ms que ahora se añade al catálogo. Un incidente bien logueado deja el sistema mejor instrumentado que antes: esa es la espiral virtuosa que justifica toda la inversión de esta guía.

FAQ rápida

¿Migro mis servicios legacy de golpe? No. Empieza por el borde (donde nace el request_id) y los servicios que más aparecen en incidentes. Los parsers del shipper pueden traducir formatos legacy mientras tanto.

¿logfmt o JSON? logfmt (clave=valor) es más legible a ojo y suficiente para estructuras planas, pero JSON gana en cuanto necesitas anidación, tipos y compatibilidad universal con shippers y plataformas. Si dudas, JSON.

¿Y en desarrollo local? Renderer legible (ConsoleRenderer en structlog, pino-pretty en Node) activado por variable de entorno. JSON en local solo castiga los ojos sin aportar nada.

¿El logging estructurado no es más lento? La serialización JSON tiene coste, pero con pino (transports en worker threads) y structlog (filtrado barato por nivel, logger cacheado) es irrelevante frente al coste de una sola consulta a base de datos. El verdadero coste del logging es la ingestión y retención, y eso se controla con muestreo, no volviendo a texto plano.

¿Cuántos campos son demasiados? La línea canónica de Stripe lleva decenas y funciona. El límite práctico no es el número sino la disciplina: campos escalares, nombres estables, tipos fijos. Un campo nuevo bien nombrado es barato; renombrar uno usado por dashboards es carísimo.

¿Guardo los logs también en un fichero local? En contenedores, no: stdout es el contrato, y la rotación es problema de la plataforma. Los ficheros locales dentro del contenedor desaparecen con el pod, llenan discos efímeros y nadie los recoge. La excepción son entornos sin orquestador (una VM clásica), donde un fichero con logrotate sigue siendo razonable.

¿Qué hago con los logs de acceso del load balancer o el ingress? Actívalos y trátalos como otra fuente estructurada más (ALB y nginx pueden emitir JSON). Son la única visión de las peticiones que murieron antes de llegar a tu aplicación —timeouts de TLS, 502 del propio proxy— y completan la historia que tus canonical lines no pueden contar.

¿Por dónde empiezo si hoy no tengo nada de esto? En una tarde: JSON en producción (structlog o pino con la configuración de esta guía), middleware de request_id, y redacción de los tres o cuatro campos sensibles obvios. Esa base ya transforma la próxima investigación. La semana siguiente: línea canónica y esquema mínimo

documentado. Todo lo demás —OTLP, muestreo fino, catálogo de eventos— se añade cuando el volumen o el equipo lo pidan, sobre una base que ya no habrá que rehacer.

Checklist de producción

- Salida JSON en producción; renderer legible solo en desarrollo.
- timestamp ISO 8601 en UTC, level, event y service en todos los logs.
- request_id generado o propagado en el borde, presente en cada línea de la petición.
- trace_id y span_id inyectados si usas OpenTelemetry.
- Logs de bibliotecas de terceros unificados en el mismo pipeline (ProcessorFormatter en Python).
- Redacción de secretos y PII configurada en el logger, no en cada llamada, y con un test que lo verifique.
- Línea canónica por petición con status, duración y campos de negocio; eventos job_started/job_finished en workers.
- Contexto (request_id, trace context) propagado dentro de los mensajes de cola.
- Esquema mínimo común documentado y alineado con las convenciones semánticas de OTel.
- Nivel info por defecto; debug activable por variable de entorno sin redespargar.
- Health checks excluidos; muestreo del ruido repetitivo; errores y líneas canónicas nunca muestreados.
- Retención por niveles y presupuesto de ingestión vigilado con una alerta de volumen.
- Logs a stdout/stderr: la rotación y el envío son trabajo de la plataforma (Docker, Kubernetes, systemd), no de tu aplicación.

Nada de esto es exótico: son dos bibliotecas maduras, un shipper bien configurado y un puñado de convenciones acordadas. La recompensa llega el día en que un pago falla en cascada entre tres servicios y reconstruyes lo ocurrido, entero, con una sola búsqueda por request_id —y la factura de la plataforma de logs no da sustos porque el ruido se quedó en el borde.

Structured Logging: JSON, Context, and Trace Correlation in Production

English edition

Turn your logs into queryable data: JSON output with structlog (Python) and pino (Node.js), per-request context with request_id, sensitive-data redaction, OpenTelemetry trace correlation, and the mistakes that make your logs useless right when you need them most. Expanded edition: ProcessorFormatter to unify third-party libraries, exceptions with context, canonical log lines, logging in workers and queues, OTLP export, sampling and cost control, targeted debug, log storms, a shared schema (OTel/ECS), Kubernetes internals, Fluent Bit and Vector, buffering and backpressure, serverless, multi-tenant, security and compliance, log testing, LogQL and CloudWatch queries, legacy-service migration, a complete hands-on incident case study, and an FAQ. With production-ready code.

Guía · Intermedio · 45 min de lectura · Julio 2026

When something breaks at three in the morning, the difference between resolving the incident in ten minutes or two hours usually comes down to your logs. If they are free-text lines, a session of `grep` and regular expressions awaits you. If they are JSON events with consistent fields, one query gives you the answer. This guide covers how to set up structured logging properly: configuring `structlog` and `pino`, per-request context, sensitive-data redaction, correlation with OpenTelemetry traces, canonical log lines, logging in workers and queues, OTLP export, cost control, shipping with Fluent Bit and Vector, testing your logs, and a complete hands-on incident investigation.

From free text to queryable events

A traditional log is a sentence written for humans: "User 42 created order ord_123 for €49.99". To find it, you need to know exactly how it was worded. A structured log is an event carrying data:

```
{
  "timestamp": "2026-07-21T09:14:03.512Z",
  "level": "info",
  "event": "order_created",
  "service": "checkout-api",
  "user_id": 42,
  "order_id": "ord_123",
  "amount_cents": 4999,
  "currency": "EUR",
  "request_id": "3f2a91c0-7b2e-4d1a-9c55-8f0f2b6a1e77"
}
```

The practical difference is enormous. With structured events you can ask "all failed orders for user 42 in the last hour" or "errors grouped by endpoint" directly in your log platform (Loki, Elasticsearch, Datadog, CloudWatch), with no brittle regex. The message stops being prose and becomes a stable event name —`order_created`, `payment_failed`— that you can count, group, and alert on.

The fields no log should omit

- timestamp in ISO 8601 and UTC. Mixing time zones across services turns any investigation into archaeology.
- level (debug, info, warning, error): filtering by severity is the first thing you will do during an incident.
- event: a short, stable snake_case name, not a sentence. "payment_failed" can be aggregated; "user X's payment failed because..." cannot.
- service and deployed version, so you know which code produced the log.
- request_id / trace_id: the identifier that ties together every log from one request across your services. Without it, correlating is guesswork.
- Domain context: user_id, order_id, job_id... the identifiers you will actually search by.

Python: structlog configured properly

`structlog` is the de facto standard for structured logging in Python. A sensible production configuration:

```
import logging
import structlog

structlog.configure(
    processors=[
        structlog.contextvars.merge_contextvars,
        structlog.processors.add_log_level,
        structlog.processors.TimeStamper(fmt="iso", utc=True),
        structlog.processors.format_exc_info,
        structlog.processors.JSONRenderer(),
    ],
    wrapper_class=structlog.make_filtering_bound_logger(logging.INFO),
    cache_logger_on_first_use=True,
```

```
)

log = structlog.get_logger()
log.info("order_created", order_id="ord_123", amount_cents=4999, currency="EUR")
# {"order_id": "ord_123", "amount_cents": 4999, "currency": "EUR",
#  "event": "order_created", "level": "info", "timestamp": "2026-07-21T09:14:03.512512Z"}
```

Each processor does exactly one thing: `merge_contextvars` pulls in the current request's context (more on that next), `TimeStamper` adds the UTC timestamp, `format_exc_info` serializes the exception when you call `log.exception()`, and `JSONRenderer` emits the final line. In development you can swap the last one for `structlog.dev.ConsoleRenderer()` and get readable, colored output without touching anything else.

Per-request context without passing the logger around

The classic mistake is dragging `request_id` as an argument through every layer of the application. Context variables solve it: you bind the data once in a middleware and it shows up in every log of that request, even inside `async` code.

```
import uuid
from structlog.contextvars import bind_contextvars, clear_contextvars

@app.middleware("http")
async def logging_context(request, call_next):
    clear_contextvars()
    request_id = request.headers.get("x-request-id") or str(uuid.uuid4())
    bind_contextvars(
        request_id=request_id,
        method=request.method,
        path=request.url.path,
    )
    response = await call_next(request)
    response.headers["x-request-id"] = request_id
    return response
```

From here on, a `log.info("stock_reserved")` in any internal service carries the `request_id` without that function knowing anything about HTTP. The initial `clear_contextvars()` prevents the previous request's context from leaking into the next one when the worker is reused.

Unifying third-party libraries: ProcessorFormatter

Your application is not the only thing that logs. `uvicorn`, `SQLAlchemy`, `boto3`, `celery`... they all use the standard library's logging module, and if you do nothing, their lines come out as plain text mixed in with your JSON. The result is a half-queryable log platform: your events yes, your dependencies' no —and the `SQLAlchemy` connection-pool `WARNING` is exactly the one you needed to group by.

`structlog` solves this with `structlog.stdlib.ProcessorFormatter`: a `logging.Formatter` that runs standard-library records through your same processor chain. Everything —your code using `structlog.get_logger()` and third-party code using `logging.getLogger()`— ends up flowing through the same pipeline, with the same JSON format and the same context fields:

```
import logging
import structlog

shared_processors = [
    structlog.contextvars.merge_contextvars,
    structlog.processors.add_log_level,
    structlog.processors.TimeStamper(fmt="iso", utc=True),
]

structlog.configure(
```

```

processors=shared_processors + [
    # converts the event_dict into something ProcessorFormatter understands
    structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
],
logger_factory=structlog.stdlib.LoggerFactory(),
cache_logger_on_first_use=True,
)

formatter = structlog.stdlib.ProcessorFormatter(
    # applied only to logs coming from logging (uvicorn, sqlalchemy...)
    foreign_pre_chain=shared_processors,
    processors=[
        structlog.stdlib.ProcessorFormatter.remove_processors_meta,
        structlog.processors.JSONRenderer(),
    ],
)

handler = logging.StreamHandler()
handler.setFormatter(formatter)
root = logging.getLogger()
root.addHandler(handler)
root.setLevel(logging.INFO)

```

Two details matter here. First, `wrap_for_formatter` must be the last processor in the structlog configuration: it places the event dictionary on the `LogRecord` so `ProcessorFormatter` renders it exactly once; without it you would end up with doubly-encoded JSON. Second, `foreign_pre_chain` applies your shared processors (timestamp, level, contextvars) to "foreign" logs too, so the `request_id` from your middleware shows up even on uvicorn's lines. Also remember to silence the handlers some libraries install on their own (uvicorn adds its own) so you don't get duplicate lines.

Exceptions with context: how to log useful errors

An error without context is a mystery; an error with context is a task. The difference between the two is decided at the moment you log it. First rule: use your logger's exception mechanism, not `str(e)`. In structlog, `log.exception()` (or `exc_info=True`) triggers `format_exc_info` and the full traceback travels serialized in a field of the event, cause chain included:

```

try:
    charge = provider.charge(order)
except ProviderTimeout as e:
    log.exception(
        "payment_failed",
        order_id=order.id,
        provider="stripe",
        amount_cents=order.amount_cents,
        attempt=attempt,
        timeout_ms=cfg.provider_timeout_ms,
    )
    raise

```

Notice which fields accompany the error: the ones someone will need to make a decision at three in the morning. Which order? Which provider? Which attempt was it? What timeout was configured? That last field is gold: it turns "it failed with a timeout" into "it failed because 5,000 ms is not enough when the provider is degraded", which is an actionable conclusion.

Second rule: log the error exactly once, at the level that handles it. The classic antipattern is catching, logging and re-raising at every layer, generating four identical tracebacks for the same failure that quadruple the noise and ruin any error count. If a layer cannot do anything with the exception, let it bubble up unlogged; the framework's global handler will record it with the request's full context.

Third: distinguish expected errors from unexpected ones. A payment declined for insufficient funds is a warning with `event="payment_declined"` —it is business, not breakage. A `NullPointerException` in the same flow is an error with a traceback. Mixing them at the same level buries real breakage under business noise, and "error rate" alerts stop

meaning anything. If you use an error aggregator like Sentry, this distinction also decides what you send it: the unexpected yes, normal business flow no.

Node.js: pino, performance, and redaction

In Node, pino is the default choice for production: it emits JSON directly and its overhead is among the lowest in the ecosystem. A base configuration with secret redaction included:

```
import pino from "pino";

export const logger = pino({
  level: process.env.LOG_LEVEL || "info",
  timestamp: pino.stdTimeFunctions.isoTime,
  base: { service: "checkout-api" },
  redact: {
    paths: [
      "req.headers.authorization",
      "req.headers.cookie",
      "/*.password",
      "/*.token",
    ],
    censor: "[REDACTED]",
  },
});

logger.info({ orderId: "ord_123", amountCents: 4999 }, "order_created");
```

The redact option deserves special attention: secrets that slip into logs end up indexed in your observability platform, out of reach of your secrets manager and its rotations. Redacting in the logger—rather than trusting every call site to be careful—is the only defense that scales.

For HTTP, pino-http creates a child logger per request with its request id:

```
import express from "express";
import pinoHttp from "pino-http";
import { randomUUID } from "node:crypto";
import { logger } from "../logger.js";

const app = express();

app.use(
  pinoHttp({
    logger,
    genReqId: (req, res) => {
      const id = req.headers["x-request-id"] || randomUUID();
      res.setHeader("x-request-id", id);
      return id;
    },
  })
);

app.post("/orders", (req, res) => {
  req.log.info({ orderId: "ord_123" }, "order_created");
  res.status(201).end();
});
```

req.log is a child logger: every line you emit with it already includes req.id and the request data, with no manual repetition.

Performance for real: transports, worker threads, and lost logs

Synchronous logging has a real cost: serializing JSON, formatting, and writing to stdout happen on the main thread, competing with your requests. pino attacks this on two fronts. First, serialization itself is brutally optimized (precompiled schemas, no dynamic inspection). Second, since v7, transports move the log destination to a worker

thread: your application serializes the line, hands it over through a shared buffer, and the worker takes care of sending it to a file, an API, or OpenTelemetry without blocking the event loop.

```
import pino from "pino";

const logger = pino({
  level: "info",
  transport: {
    target: "pino-opentelemetry-transport", // runs in a worker thread
  },
});
```

This architecture has a trade-off you must know about: if the process dies abruptly (OOM kill, native crash), the lines sitting in the transport's buffer can be lost—including, with bad luck, the ones explaining why it died. For critical flows, pino offers `pino.destination({ sync: true })` and transport-level `flushSync`; and in any case you should catch termination signals and call `logger.flush()` before exiting.

In Python the balance is different: `structlog` is fast for what it does (the practical recommendation is enabling `cache_logger_on_first_use=True` and using `make_filtering_bound_logger`, which discards disabled levels almost for free), but writing remains synchronous. If volume becomes a problem, the answer is usually not an exotic async logger but writing less: lowering levels, sampling, and aggregating. And one universal rule in both languages: never do expensive work to build arguments for a log that may be disabled—no `json.dumps(huge_object)` as a debug argument.

Correlating with OpenTelemetry traces

If you already use distributed tracing, your logs should carry the `active_trace_id` and `span_id`. That is the bridge between both signals: from a slow trace you jump to its exact logs, and from an error log you open the full trace that produced it. In `structlog`, a small custom processor is all it takes:

```
from opentelemetry import trace

def add_trace_context(logger, method_name, event_dict):
    span = trace.get_current_span()
    ctx = span.get_span_context()
    if ctx.is_valid:
        event_dict["trace_id"] = format(ctx.trace_id, "032x")
        event_dict["span_id"] = format(ctx.span_id, "016x")
    return event_dict

# add it to the processors list, before JSONRenderer
```

In Node you do not even need to write it: the `@opentelemetry/instrumentation-pino` package injects `trace_id` and `span_id` into every line automatically whenever a span is active.

OpenTelemetry Logs: OTLP export without marrying anyone

So far we have talked about writing JSON to stdout and letting the platform pick it up. OpenTelemetry proposes one step further: treating logs as a first-class signal, with its own data model (LogRecord with attributes, resource, and native trace context) and its own export protocol (OTLP). The strategic advantage is neutrality: the same pipeline works for Loki, Elasticsearch, Datadog, or any compatible backend, and switching vendors means changing the Collector's configuration, not your code.

In Python, the bridge is built from the standard logging module with a `LoggingHandler` that translates each `LogRecord` into the OTel model and sends it in batches:

```

from opentelemetry._logs import set_logger_provider
from opentelemetry.exporter.otlp.proto.grpc._log_exporter import OTLPLogExporter
from opentelemetry.sdk._logs import LoggerProvider, LoggingHandler
from opentelemetry.sdk._logs.export import BatchLogRecordProcessor
from opentelemetry.sdk.resources import Resource
import logging

provider = LoggerProvider(
    resource=Resource.create({"service.name": "checkout-api"})
)
provider.add_log_record_processor(
    BatchLogRecordProcessor(OTLPLogExporter(endpoint="http://otel-collector:4317"))
)
set_logger_provider(provider)

handler = LoggingHandler(level=logging.INFO, logger_provider=provider)
logging.getLogger().addHandler(handler)

```

An honest caveat: the OTel logs specification is stable, but in the Python SDK these modules still live under the `_logs` namespace, a sign that the API can still change between minor versions. Pin your versions and read the changelogs before upgrading.

In Node, `pino-opentelemetry-transport` does all the work inside a worker thread: it starts an OTel `LoggerProvider` in the worker, parses each line pino emits, translates it into the OTel Logs data model, and sends it over OTLP. Your main thread only serializes JSON, as always. A very reasonable adoption pattern is hybrid: keep writing to stdout (your safety net, always readable with `kubectl logs`) and add OTLP as a second destination from the Collector, not from the application.

Canonical log lines: one wide event per request

There is a technique that multiplies the value of your logs without multiplying their volume: the canonical log line, popularized by Stripe. The idea is that, in addition to the point-in-time logs you emit during a request, every request ends with a single "wide" event summarizing everything that happened: who called, which endpoint, with what result, how long each dependency took, which business decisions were made.

```

import time
from structlog.contextvars import bind_contextvars

@app.middleware("http")
async def canonical_line(request, call_next):
    start = time.perf_counter()
    response = await call_next(request)
    log.info(
        "http_request",          # the canonical event, one per request
        status=response.status_code,
        duration_ms=round((time.perf_counter() - start) * 1000, 1),
        # business fields accumulated in contextvars during the request:
        # user_id, plan, cache_hit, db_queries, payment_provider...
    )
    return response

# in any layer of the application, without touching signatures:
def apply_discount(order):
    bind_contextvars(discount_code=order.discount, discount_pct=15)

```

The practical consequence: 90% of incident questions ("which enterprise-plan requests failed with a timeout at the payment provider?") are answered by querying a single event type with many fields, instead of reconstructing the story by stitching together eight scattered lines. Point-in-time logs still exist for the detail; the canonical line is the index. If you can only afford to retain part of your logs, retain these.

Two implementation tips: emit the canonical line also when the request fails (the middleware must catch the exception, annotate `error_type`, and re-raise it), and resist the temptation to include whole payloads: scalar fields with stable names, not ten-level nested documents.

Workers and queues: the request_id must not die at the broker

Everything above works beautifully... until the request enqueues a job and the context evaporates. The consumer processes the message half an hour later, in another process on another machine, and its logs no longer carry request_id or trace_id. The rule is simple: context travels inside the message, just as it would travel in an HTTP header.

```
# Producer: attach the context when enqueueing
queue.publish({
    "type": "send_invoice",
    "order_id": "ord_123",
    "_ctx": {"request_id": request_id}, # the context travels with the message
})

# Consumer (Celery, RQ, Kafka consumer...): re-bind when processing
from structlog.contextvars import bind_contextvars, clear_contextvars

def handle(message):
    clear_contextvars()
    bind_contextvars(
        request_id=message.get("_ctx", {}).get("request_id"),
        job_type=message["type"],
        job_id=message["id"],
    )
    log.info("job_started")
    ...
    log.info("job_finished", duration_ms=elapsed)
```

In Node the pattern is identical with BullMQ: store request_id in job.data when enqueueing and create a child logger in the worker with `logger.child({ requestId: job.data.requestId, jobId: job.id })`. If you use OpenTelemetry, also propagate the serialized trace context (W3C propagators do this for you in many integrations) so the trace crosses the broker.

The job_started / job_finished / job_failed events with job_id, attempt, and duration are the asynchronous version of the canonical line: they let you answer "how many retries happened last night, and of what kind?" without leaving the log platform. And when a job fails after exhausting its retries, that final log must carry the original request_id: it is what connects this morning's user complaint with the job that died overnight.

Levels: what to log (and what not to)

In production, info is the sensible default level. Relevant business events —signup completed, order created, payment declined, deploy applied— go to info; anomalous but handled situations (a retry, a fallback kicking in) go to warning; and error is reserved for failures someone should actually look at. debug stays off unless you are actively investigating: the ingestion cost and the noise are not worth keeping it always on.

Two rules save a lot of money and noise: do not log inside hot loops (aggregate and emit a summary at the end), and do not use logs as metrics. Counting requests or measuring latencies is a job for counters and histograms; logs provide the detail of specific cases.

Sampling and cost control: logging less without seeing less

Your log bill grows with traffic, not with value. A healthy service at scale emits millions of identical "all good" lines that you pay to ingest, index, and retain. The control techniques, from simplest to finest:

- Drop at the source what you will never query: health checks (GET /healthz every 5 seconds × N pods), redirects, CORS preflight requests. A filter in the shipper or an if in the middleware.
- Sample the repetitive and successful: keeping 1-10% of the info logs on hot paths preserves statistical visibility at a fraction of the cost. Configured well, sampling and throttling can cut volume by 70-90% without losing

diagnostic capability.

- Never sample errors or canonical lines: the golden rule. warning and error always pass; the canonical log line always passes. What gets sampled is background noise, not signal.
- Retention by level: errors for 90 days and info for 7 costs a fraction of "everything for 90 days". Most platforms support this natively.
- Dynamic level without redeploying: being able to raise one specific service to debug for twenty minutes of investigation (environment variable, admin endpoint, or SIGHUP) and lower it again. It is the cheap alternative to keeping debug always on "just in case".

A useful heuristic for deciding what to sample: if you deleted 99% of these lines, would you lose any incident-response capability? For "request completed 200 OK on /api/products" the answer is no —latency and counts already live in your metrics, and the rare detail is covered by the canonical line.

Targeted debug: surgical verbosity in production

Between "debug off" and "debug for everyone" there is a middle ground far superior to both: enabling verbosity only for the request, user, or tenant you are investigating. The mechanism is simple if you already have per-request context: a header or a flag (for example, x-debug-request: 1, emitted only by internal tools, or a feature flag per user_id) that the middleware detects and uses to lower the level threshold for that specific request.

```
@app.middleware("http")
async def targeted_debug(request, call_next):
    debug_users = await flags.get("debug_user_ids")    # feature flag
    user_id = get_user_id(request)
    if user_id in debug_users:
        bind_contextvars(debug_session=True)
        # this request's logger drops to DEBUG, and only here
    return await call_next(request)
```

The result: the user who has spent three days reporting an unreproducible bug generates debug-level logs with full detail, while the rest of the traffic stays at info and the bill barely notices. The same pattern works for a canary: pods running the new version start with extra verbosity for the first hour. Two precautions: debug flags must expire on their own (a TTL on the flag, not a mental note), and debug level is still no excuse to dump payloads with PII —the logger's redaction applies all the same.

Metrics from logs: where the boundary lies

Modern platforms let you derive metrics from logs (log-based metrics in Cloud Logging, recording rules over LogQL in Loki, log-to-metrics in Datadog), and the temptation to use logs as the universal source is real. The healthy boundary: native metrics (counters and histograms instrumented in code) are the source of truth for rates, latencies, and saturation —they are cheap, precise, and do not depend on the log pipeline. Log-derived metrics shine for what you did not instrument in time: counting a specific error_type during an incident, measuring the frequency of a newly discovered business event, validating a hypothesis before instrumenting it properly.

The trap to avoid is structural dependency: if your SLO is computed from logs, any change in format, sampling, or retention alters your SLO without anyone touching the service. Sampling 90% of the noise —perfectly reasonable for diagnostics— is devastating for a derived metric that assumed it saw every line. Practical rule: metrics feeding alerts and SLOs are instrumented in code; log-derived ones are exploration tools and temporary bridges, and canonical lines (never sampled) are the only acceptable basis when there is no other choice.

Log storms: when the same error repeats a hundred thousand times

There is a failure mode no static sampling anticipates: the downed dependency that turns every request into three error lines and a traceback. Within minutes, the service goes from 50 lines per second to 20,000, the shipper

chokes, the platform applies rate limiting and —here is the cruel part— the logs the rate limit drops are exactly the ones from the incident you are trying to investigate. A log storm is an incident inside the incident.

The defenses are familiar from other contexts and apply the same way here. Windowed deduplication: if the same (event, error_type, service) repeats, emit the first occurrence in full and then a periodic summary ("this error occurred 4,812 more times in the last 60 s") instead of every instance —zerolog's native sampling and equivalent utilities in other ecosystems do this almost for free. Per-logger rate limiting: a cap of N errors per second per event type, applied in the application or in the shipper (throttle in Fluent Bit, stateful VRL in Vector). And the structural version: if the error comes from a dependency, the circuit breaker that protects your latency also protects your logs, because once it opens it stops generating a failure (and its three lines) per request, replacing them with a single circuit_opened event.

The acid test is cheap: in staging, take down a dependency on purpose and watch how many lines per second your service emits. If the answer looks expensive multiplied by your production pods and an hour of incident, you know which defense is missing. It is the kind of discovery you would rather make on a Tuesday afternoon than during the real outage.

A shared schema: semantic conventions

Structured logging only pays off if fields mean the same thing across all services. If team A writes `userId`, team B `user_id`, and team C `uid`, every cross-service query needs three variants and your dashboards fill up with ORs. The solution is not heroic: a one-page document with the minimal shared schema, ideally aligned with an existing standard.

The two standards that matter are OpenTelemetry's semantic conventions (service.name, http.request.method, http.response.status_code, url.path...) and ECS, the Elastic Common Schema —which since 2023 has been officially converging with OTel's conventions, so betting on OTel is betting on the meeting point. You do not need to adopt the whole catalog: pick the dozen fields that cross services (identifiers, HTTP, errors) and leave freedom in the rest.

```
// Agreed minimal schema (example)
{
  "timestamp": "...",          // ISO 8601 UTC, always
  "level": "info",             // debug | info | warning | error
  "event": "order_created",    // snake_case, documented catalog
  "service": "checkout-api",   // canonical service name
  "version": "2026.07.15",     // deployed version
  "request_id": "...",         // propagated from the edge
  "trace_id": "...",          // if tracing is on
  "user_id": 42,               // numeric, not "user-42"
  "error.type": "TimeoutError", // errors only
  "error.message": "...",      // errors only
}
```

Two details that save you grief: also pin the types (a `user_id` that is sometimes a number and sometimes a string breaks Elasticsearch's typed indexes), and version the schema in a repository with change review, because renaming a field queried by twenty dashboards is an incident in its own right.

Kubernetes internals: what happens between your stdout and the shipper

Writing to stdout is half the story; it pays to know what happens next, because several production "mysteries" live there. The container runtime (containerd, CRI-O) captures each container's stdout and writes it to a file on the node (`/var/log/pods/...`) in CRI format: each of your lines gets wrapped with a timestamp, the stream, and a marker for whether the line was split. And here comes the first subtlety: the runtime splits lines that exceed its limit (16 KB in containerd by default), so a giant JSON event can arrive chopped up. Shippers know how to reassemble it (the CRI parsing option in Fluent Bit), but it is one more reason not to log huge payloads.

Second subtlety: rotation. The kubelet rotates these files by size (containerLogMaxSize, 10 MB by default) and

keeps only a few rotations. If your service emits massive bursts —an error in a loop, say— rotation can outrun the shipper and those lines are lost without a trace. The defense is twofold: do not log in loops (a logging circuit breaker, or at least an aggregated counter) and size the shipper's buffer with headroom.

Third: `kubectl logs` reads those same node files, not your platform. That is why it works even when the observability pipeline is down —it is your safety net— but also why it only shows what has not yet rotated, and nothing from already-deleted pods. For the full story, the platform; for the immediate look at the pod that just crashed on startup, `kubectl logs --previous` is often the fastest tool there is.

From stdout to your platform: Fluent Bit, Vector, and the Collector

Your application writes JSON to stdout and its responsibility ends there. The journey to the log platform is handled by a shipper, and in Kubernetes the dominant pattern is a `DaemonSet` that reads each node's container files, enriches every line with metadata (pod, namespace, labels), and sends it to the backend. The three serious options:

- **Fluent Bit**: the lightweight standard (20-30 MB of memory per node, CNCF pedigree). It parses, filters, enriches via its Kubernetes filter, and ships to practically any backend. For most clusters it is the default choice.
- **Vector**: written in Rust, more capable at transformations (its VRL language lets you reshape, redact, and sample with real logic) and with guaranteed-delivery semantics via end-to-end acknowledgements. It consumes more (50-80 MB typical) and has one known edge: if a pod is deleted while Vector still has a backlog of its logs, that pod's Kubernetes metadata can be lost.
- **OpenTelemetry Collector**: the natural option if you already use it for traces and metrics —a single agent for all three signals, with the `filelog` receiver for container logs.

```
# Fluent Bit: parse container JSON and sample noise (excerpt)
[FILTER]
    Name    kubernetes
    Match   kube.*
    Merge_Log On          # the app's JSON becomes first-class fields

[FILTER]
    Name    grep
    Match   kube.*
    Exclude $path /healthz # health checks do not travel

[OUTPUT]
    Name    loki
    Match   kube.*
    labels  job=fluentbit, $kubernetes['namespace_name']
```

Two tips with any shipper. First, beware high-cardinality labels/indexes: in Loki, every label combination creates a stream —label by namespace and service, never by `request_id` or `user_id` (those are found with filters, not labels). Second, resolve multiline stack traces at the edge if some application still does not emit JSON: Fluent Bit's multiline parsers exist precisely because a Java traceback split into 40 loose logs is useless.

When the pipeline goes down: buffering, backpressure, and delivery

The log pipeline fails too: the platform has an incident, the network congests, the backend applies rate limiting. What happens then depends on decisions best made calmly, in advance. The first is the shipper's buffer: in memory (fast, lost if the agent dies) or on disk (survives restarts, consumes node I/O). For application logs, a hybrid buffer with disk overflow and a clear limit is the usual balance; for audit logs, disk always. The second is the overflow policy when the buffer fills: drop the oldest, drop the newest, or apply backpressure? Fluent Bit and Vector let you choose, and the right answer depends on the stream —dropping old debug is acceptable; dropping recent errors is not.

The third decision is what end-to-end guarantee you want. Vector offers end-to-end acknowledgements: the agent does not consider a line delivered until the backend confirms it, turning "I think it arrived" into "it arrived". The price

is more memory and the possibility of backpressure: if the backend slows down, the queue grows backwards. What matters is that backpressure never reaches your application —writing to stdout must never block because of the pipeline; that isolation is precisely the architectural reason the shipper is a separate process and not a library inside your app.

And the loop closes with observability of the pipeline itself: the shipper exports metrics (lines read, sent, dropped, buffer size, retries) and they deserve their own alert. A steadily growing buffer warns of a degraded backend hours before the first line is lost; a non-zero drop counter during an incident explains why "logs are missing" in the follow-up investigation. The question "can we trust that last night's logs are complete?" must have a verifiable answer, not a hopeful one.

Serverless: logging with no daemon to save you

On AWS Lambda, Cloud Run, or Azure Functions there is no DaemonSet or sidecar: stdout goes straight to the platform's log service (CloudWatch Logs, Cloud Logging), and your only lever is what you write. The rules change a bit. First: structured JSON matters even more, because CloudWatch Logs Insights and Cloud Logging discover JSON fields automatically and give you typed queries for free; Lambda even supports a native JSON log format, configurable per function, that structures the runtime's own logs too.

Second: the invocation context is your `request_id`. In Lambda, `context.aws_request_id` (and the X-Ray or OTel trace id if you use one) must be bound into every line just as you would in an HTTP middleware —same contextvars pattern, different entry point. Third: watch the flush. The environment freezes as soon as the handler returns; buffered async transports (including OTLP's) can lose the last lines if you do not force a flush before returning. In serverless, synchronous writes to stdout are not a limitation: they are the correct choice.

And an economic warning: CloudWatch charges for ingestion at a price that turns a forgotten debug in your hottest function into a visible line item on the monthly bill. The techniques from the cost section —info as the default level, sampling the noise, short retention for the verbose— apply here with extra force, and per-log-group retention is one click away.

Beyond Python and Node?

The patterns in this guide are language-agnostic; only the libraries change. In Go, the standard library itself has included the log/slog package since 1.21, with JSON output, typed attributes, and composable handlers —for new services there is no excuse left for `log.Printf`— and zerolog remains the reference when performance is critical. In Java, the usual combination is SLF4J/Logback with logstash-logback-encoder for JSON and MDC for per-request context (the equivalent of contextvars). In .NET, Serilog with enrichers plays the same role, and Microsoft.Extensions.Logging supports structured scopes out of the box. In Rust, tracing unifies logs and spans in a single model, which fits this guide's correlation philosophy natively.

The practical consequence for polyglot teams: the shared schema document (fields, types, event names) weighs more than any library choice. If the Go service calls the field `user_id` and the Java one `userId`, you will have reproduced the very problem structured logging came to solve, with perfectly valid JSON on both sides.

Multi-tenant: `tenant_id` is a first-class field

If your product serves multiple customers from the same infrastructure, `tenant_id` deserves the same treatment as `request_id`: bound at the edge (from the token or the subdomain), carried through contextvars, and present on every line, including the workers'. The benefits are immediate: "does this incident affect all customers or just one?" is the first question on any SaaS on-call shift, and with `tenant_id` on the canonical line it is answered with a group by. It also enables per-customer cost analysis (which tenant generates 40% of the log volume?) and targeted support: the surgical debug from the earlier section applied to one specific tenant during an investigation.

There are two precautions. On label-indexed platforms like Loki, `tenant_id` as a label is only viable with few tenants

(dozens); with thousands, it is a filterable field, not a label —cardinality again. And if your contracts include data isolation or residency, logs count as customer data: per-tenant routing in the shipper (separate streams or buckets) is the natural place to enforce it, one more reason for `tenant_id` to be a reliable field rather than a half-followed convention.

Security and compliance: beyond redaction

Redaction in the logger is the first line of defense, not the last. A complete program has four layers. First: classify what counts as PII in your domain (emails, IPs, names, addresses, tokens, and combinations that identify indirectly) and decide field by field: is it logged, redacted, or transformed? Second: transform at the source when you need to correlate without exposing —a keyed HMAC of the email (`user_hash`) lets you group all of one user's logs without storing the email; truncating the IP (192.168.1.0/24) is usually enough for geographic analysis. Third: a safety net in the shipper (VRL in Vector, lua filters in Fluent Bit, or the Collector's transform processor) that catches card-number, email, or token patterns that slipped through despite everything. Fourth: retention and access —logs containing personal data carry the same GDPR obligations as your database, but without its selective-deletion tooling, so short retention is your most realistic compliance mechanism.

Audit logs deserve separate mention (who accessed what, permission changes, administrative actions): they have requirements opposite to application logs —long retention, immutability, restricted access— and are worth separating into their own stream from day one, not fishing out of the operational noise afterwards.

Logs get reviewed too: the event as a team contract

A living event catalog is what separates "structured logging" from "structured JSON with chaos inside". It works like the rest of the service's contract: event names and their fields live in a versioned document (a page in the repository is enough), and adding or changing an event goes through PR review like any API change. The review need not be bureaucratic; three questions suffice. Does the name follow the convention (`snake_case`, past-tense verb, no identifiers inside)? Do the new fields reuse names and types from the shared schema? Does this log add anything that metrics and existing events do not already cover?

That third question is the most valuable, because every team's natural impulse is to log more "just in case", and every new line is perpetual ingestion cost and noise. The reverse discipline applies too: when an event stops being queried (platforms usually show usage statistics for fields and saved queries), it gets deprecated and removed. A catalog of forty well-named events everyone knows is worth more than four hundred nobody remembers. And a side benefit that shows up during onboarding: the event catalog is, in practice, a map of what the system does —reading it is one of the fastest ways to understand a new service.

A natural companion to the catalog: log your configuration at startup. A single `service_started` event with the deployed version, active flags, and relevant configuration values (redacted where appropriate) turns the question "what configuration did the pod have when this happened?" into a one-second search, and it is the first clue when a deploy changes behavior without changing code.

Logging in LLM applications: tokens, cost, and privacy

If your service calls a language model, your logs have a new job: explaining what the user asked, which model answered, how much it cost, and why it failed. An LLM call is an expensive, slow, non-deterministic external dependency — exactly the kind of operation that most justifies one structured event per invocation. And yet most teams that already log their HTTP requests properly treat model calls as a black box that leaves behind nothing but a free-text INFO Calling OpenAI....

OpenTelemetry already defines semantic conventions for generative AI: the `gen_ai.*` attributes cover the provider, the requested model and the actual response model, input and output tokens, and the finish reason. They are still marked experimental — the work is mid-flight, and in 2026 it moved to its own repository — but the major observability vendors already support them. Adopting their names today (`gen_ai.request.model`,

gen_ai.usage.input_tokens, gen_ai.usage.output_tokens) saves you a field-renaming project tomorrow, even if you use nothing else from OTel.

The minimum event per call should answer four questions: which model, with which parameters; how many tokens, at what cost; how long it took; and how it ended (natural stop, token limit, content filter, or error). Here is what that looks like with structlog:

```
import hashlib
import time

import structlog

log = structlog.get_logger()

# Per-token model pricing (keep it in config, not hardcoded)
IN_PRICE = 3.0 / 1_000_000 # USD per input token
OUT_PRICE = 15.0 / 1_000_000 # USD per output token

async def call_llm(client, *, model, messages, **params):
    prompt_text = "".join(m["content"] for m in messages)
    prompt_hash = hashlib.sha256(prompt_text.encode()).hexdigest()[:16]
    start = time.monotonic()
    try:
        resp = await client.chat.completions.create(
            model=model, messages=messages, **params
        )
    except Exception as exc:
        log.error(
            "llm_call_failed",
            model=model,
            prompt_hash=prompt_hash,
            prompt_chars=len(prompt_text),
            duration_ms=int((time.monotonic() - start) * 1000),
            error_type=type(exc).__name__,
        )
        raise

    usage = resp.usage
    log.info(
        "llm_call",
        **{
            "gen_ai.request.model": model,
            "gen_ai.response.model": resp.model,
            "gen_ai.usage.input_tokens": usage.prompt_tokens,
            "gen_ai.usage.output_tokens": usage.completion_tokens,
        },
        finish_reason=resp.choices[0].finish_reason,
        prompt_hash=prompt_hash,
        prompt_chars=len(prompt_text),
        duration_ms=int((time.monotonic() - start) * 1000),
        cost_usd=round(
            usage.prompt_tokens * IN_PRICE
            + usage.completion_tokens * OUT_PRICE, 6
        ),
    ),
    return resp
```

Notice what the event does not include: neither the raw prompt nor the raw response. Prompts contain user data at an extremely high rate — names, emails, medical histories, contracts — and a general-purpose log with 30-day retention is the worst possible place to store them. The hash and the length are enough to group identical calls, spot abnormally long prompts, and correlate retries. If you truly need to inspect payloads (to debug quality or build evals), capture them sampled and redacted into a separate store with short retention and restricted access, not into the shared logging pipeline.

The `finish_reason` deserves a special mention: it is the field that explains the most quality incidents and the one almost nobody logs. A spike in length means responses truncated by the token limit; a spike in `content_filter` means the provider is blocking outputs. Both are invisible if all you measure is latency and HTTP errors.

For agents, the canonical log line pattern we saw earlier is a perfect fit: besides the per-call event, emit one wide

event per complete agent run, with accumulated totals:

```
{
  "event": "agent_run_completed",
  "run_id": "run_9f2c",
  "request_id": "3f2a91c0-7b2e-4d1a-9c55-8f0f2b6a1e77",
  "gen_ai.request.model": "claude-sonnet-5",
  "steps": 4,
  "tool_calls": ["search_db", "send_email"],
  "input_tokens_total": 8210,
  "output_tokens_total": 1467,
  "cost_usd": 0.041,
  "duration_ms": 9302,
  "outcome": "success"
}
```

With that event, questions like "how much does the support agent cost us per day", "what percentage of runs exhaust the step limit", or "which tool fails most often" become an aggregation query instead of an investigation. And when product asks why the provider bill jumped 40% this month, the answer will live in a sum by (`gen_ai.request.model`) rather than in a hand-built spreadsheet.

From the browser to the backend: correlation doesn't start at your API

A user reports "it doesn't work", you open the backend logs, and everything is green. The failure happened in the browser: a fetch the ad blocker killed, a JavaScript exception, a timeout the client never retried. If your observability starts at the load balancer, the first half of every request is uncharted territory.

Closing that gap has two pieces. The first is propagating trace context from the browser: if the frontend's fetch sends the W3C Trace Context traceparent header, the `trace_id` your backend already injects into every log (as we saw in the OpenTelemetry section) connects the user's action to every service it touched. OpenTelemetry's JavaScript auto-instrumentations do this on their own; if you don't want to ship the full SDK, generating it by hand is ten lines:

```
// Propagate traceparent (W3C Trace Context) on every fetch
function traceparent() {
  const hex = (n) =>
    Array.from(crypto.getRandomValues(new Uint8Array(n)))
      .map((b) => b.toString(16).padStart(2, "0"))
      .join("");
  return "00-" + hex(16) + "-" + hex(8) + "-01";
}

const _fetch = window.fetch;
window.fetch = (url, opts = {}) => {
  opts.headers = { ...(opts.headers || {}), traceparent: traceparent() };
  return _fetch(url, opts);
};
```

The second piece is shipping browser events as structured logs. A decent client-side logger accumulates events in memory and sends them in batches with `navigator.sendBeacon`, which survives the tab being closed:

```
const buffer = [];

export function clientLog(event, fields = {}) {
  buffer.push({
    event,
    ts: Date.now(),
    url: location.pathname,
    session_id: sessionId,
    ...fields,
  });
  if (buffer.length >= 20) flush();
}
```

```
function flush() {
  if (!buffer.length) return;
  navigator.sendBeacon("/client-logs", JSON.stringify(buffer.splice(0)));
}

addEventListener("visibilitychange", () => {
  if (document.visibilityState === "hidden") flush();
});

window.addEventListener("error", (e) =>
  clientLog("js_error", {
    message: e.message,
    source: e.filename,
    line: e.lineno,
  })
);
```

On the backend, the `/client-logs` endpoint validates the batch, enriches it with what the browser doesn't know or could fake (truncated IP, parsed user agent, the authenticated session's `user_id`), and emits it through the same pipeline as the rest of your logs, with service: "frontend". If you'd rather not build it, SDKs like Grafana Faro package exactly this pattern: error capture, Web Vitals, and logs with trace correlation included.

The typical mistakes in this integration are well known and avoidable. Serve the endpoint from your own domain with a neutral name — routes like `/collect` or `/analytics` die at the hands of ad blockers. Sample aggressively: a thousand open tabs emitting scroll events can generate more volume than your entire API. Truncate stack traces before sending them. And never log the full URL with its query params: invitation tokens, OAuth codes, and personal identifiers live there at an alarming rate.

Alerting on logs: from the pretty dashboard to the 3:07 a.m. page

A queryable log that nobody queries until a customer complains is wasted potential. The last piece of the pipeline is alerting on the logs themselves, and it's where the work of structuring every event pays for itself: alerting on bare `level="error"` is impossible to tune, but alerting on the rate of `payment_failed` relative to total payment attempts is a business signal.

In the Grafana ecosystem, the Loki Ruler evaluates LogQL queries on a schedule and sends the resulting alerts to Alertmanager, exactly like Prometheus does with metrics. A reasonable rule for the payments case:

```
groups:
- name: checkout-alerts
  rules:
  - alert: PaymentErrorRateHigh
    expr: |
      sum(rate({app="checkout-api"} | json
        | level = "error" | event =~ "payment_.*" [5m]))
      /
      sum(rate({app="checkout-api"} | json
        | event =~ "payment_.*" [5m]))
      > 0.05
    for: 10m
    labels:
      severity: page
    annotations:
      summary: "More than 5% of payment events failing"
      runbook: "https://wiki.internal/runbooks/payments"
```

The principles that make a log-based alert work are the same as with metrics, with a couple of log-specific twists. Alert on ratios, not absolute counts: five errors per minute is catastrophic at ten requests and noise at ten thousand. Use `for` to require the condition to persist so nobody gets woken up by a thirty-second blip. And alert on symptoms a user would notice (payments failing, p99 latency, queues growing), not on internal causes that may have no impact — causes belong on the diagnostic dashboard, not on the pager.

The log-specific twist is cardinality: alerting per tenant, per customer, or on a rare combination of fields —

prohibitively expensive as a Prometheus metric because it explodes the number of series — is almost free as a log query. If a LogQL rule becomes too costly to evaluate every minute, turn it into a recording rule: the Ruler materializes it as a metric via remote write, and the alert is then evaluated against Prometheus, which is cheap at exactly that job.

What a log costs: budget before surprise

Let's end with the uncomfortable conversation. Structured logging multiplies the value of each event, but it also makes it very easy to emit many more of them, and the observability bill is famously the year's second surprise after the LLM provider's. The defense is arithmetic done beforehand, not afterwards:

```
200 req/s × 3 events/req × 600 bytes ~ 360 KB/s
360 KB/s × 86,400 s ~ 31 GB/day
31 GB/day × 30 days of retention ~ 930 GB stored
930 GB × (ingest + retention pricing) = your monthly bill
```

With those numbers in front of you, the levers from earlier sections stop being theory: sampling success logs shrinks the first factor; canonical log lines shrink the second (one wide event instead of eight narrow ones); not logging payloads shrinks the third. Tiered retention attacks the fourth: a few days in hot, instantly-queryable storage, a few weeks in cheap object-storage cold tiers, and archive only what compliance demands. Most incident queries look at the last few hours; paying premium storage for ninety-day-old logs is paying for a Ferrari to drive to the mailbox.

Two team practices close the loop. First: a volume budget per service, visible on a dashboard, with the same status as the latency budget — a deploy that triples emitted events should stand out in review exactly like one that triples p99. Second: filtering at the front door — Vector or Fluent Bit dropping forgotten debug events, health checks, and bot traffic before they reach ingestion — is the optimization with the best effort-to-savings ratio in the whole pipeline. The goal is never to log less for its own sake: it's for every gigabyte you pay for to answer questions someone actually asks.

Evolving the schema without breaking the dashboards

The event catalog from the previous section solves the present; this one solves the future. Every log schema changes — fields get renamed, types get corrected, events get split — and every change has silent consumers: dashboards, Ruler alerts, saved queries, and that finance script nobody remembers. Breaking them throws no error in any deploy; a panel just flatlines to zero and an alert quietly stops firing, which is the worst possible way to fail.

The rules that prevent that scenario are the same as for any public API. Additive changes — adding a new field — are always safe and need no ceremony. Breaking ones — renaming user to user_id, changing amount from string to number, splitting checkout_done into two events — demand a dual-emission window: the service emits the old and the new field at the same time, you migrate consumers calmly, and only then retire the old one. Two weeks of a duplicated field are infinitely cheaper than a payments alert that's blind during an incident.

For critical events — the ones feeding alerts that wake someone up or reports that leave the team — it's worth pinning the contract in CI, just as you would with an API schema. A test that validates emitted events against the catalog turns "someone renamed a field without telling anyone" from an incident into a build failure:

```
import jsonschema

PAYMENT_FAILED_V1 = {
    "type": "object",
    "required": [
        "event", "request_id", "order_id",
        "amount_cents", "error_code",
    ],
    "properties": {
        "event": {"const": "payment_failed"},
    },
}
```

```

    "amount_cents": {"type": "integer"},
    "error_code": {"type": "string"},
  },
}

def test_payment_failed_matches_contract(capture_logs):
    trigger_failed_payment()
    event = capture_logs.find(event="payment_failed")
    jsonschema.validate(event, PAYMENT_FAILED_V1)

```

If an event must change incompatibly and dual emission isn't viable, an `event_version` field is the honest plan B: `payment_failed` with `event_version: 2` lets queries tell the eras apart and lets dashboards show the seam instead of hiding it. What doesn't work is silent mutation: the same event name with two different shapes depending on the deploy date is a trap for the teammate investigating an incident six months from now.

The cheap version of all this, if your team is small: before merging a change that touches an existing event's fields, search for its name in the saved queries and alert rules. Five minutes of grep in the dashboards repo prevents most breakage, with no new infrastructure.

How to test your logs

Logs are one more interface of your system —alerts and runbooks depend on them— and they deserve tests like any interface. You do not need to test every line: just the events that alerts, dashboards, or audits depend on. `structlog` makes it easy with `capture_logs`:

```

from structlog.testing import capture_logs

def test_failed_payment_emits_event():
    with capture_logs() as logs:
        process_payment(order_with_bad_card)

    events = [l for l in logs if l["event"] == "payment_failed"]
    assert len(events) == 1
    assert events[0]["order_id"] == "ord_123"
    assert events[0]["log_level"] == "warning"
    assert "card_number" not in events[0]          # redaction gets tested too

```

In Node, the equivalent pattern is injecting a test stream into `pino` and asserting on the parsed objects:

```

import pino from "pino";

test("failed payment emits event", () => {
  const lines = [];
  const logger = pino({ level: "info" }, {
    write(chunk) { lines.push(JSON.parse(chunk)); },
  });

  processPayment(orderWithBadCard, logger);

  const evt = lines.find((l) => l.msg === "payment_failed");
  expect(evt.orderId).toBe("ord_123");
  expect(evt.cardNumber).toBeUndefined(); // redacted
});

```

The most valuable test of all is the one verifying that redaction works: it is cheap, runs on every CI, and the alternative is discovering in an audit that you had been indexing card numbers for six months. Add a schema smoke test too (every event carries timestamp, level, event, service) and you will have armored the contract your dashboards assume.

Common mistakes that cost you dearly

- Double serialization: passing already-serialized JSON as the message produces JSON inside JSON, impossible to query. Always pass objects and fields, never preformatted strings.
- Uncontrolled cardinality in event names: "user_42_order_failed" creates a new event per user. The identifier goes in a field; the event name stays fixed.
- PII in logs: emails, tokens, card numbers. Beyond the security risk, it complicates compliance (GDPR): an indexed log cannot easily be "selectively deleted". Redact at the source.
- Multi-line stack traces as plain text: each line arrives at the platform as a separate log. With JSON, the full exception travels in a field of the same event.
- Different formats per service: if each team names fields its own way (userId, user_id, uid), cross-service queries die. Agree on a minimal shared schema and document it.
- Trusting that "we'll remove it later": debug console.log and print statements always reach production. A linter that bans them (no-console in ESLint, flake8-print) takes five minutes and prevents it forever.
- Inconsistent types in the same field: a duration_ms that is sometimes 123 and sometimes "123ms" breaks aggregations and, in backends with typed indexes, silently drops whole documents. Pin the types in the schema.
- Logs as the only signal: if your only way of knowing the service is unhealthy is searching for errors in the logs, you discover incidents late. Metrics alert; logs explain; traces localize.

Queries worth mastering

Structured logging pays for itself in the query language. It is worth practicing three or four patterns before the incident, not during it. In Loki (LogQL), the typical pipeline is selecting by labels, parsing JSON, and filtering by fields:

```
# errors grouped by type over the last 30 min
sum by (error_type) (
  count_over_time(
    {service="checkout-api"} | json | level="error" [30m]
  )
)

# p95 latency from the canonical lines
quantile_over_time(0.95,
  {service="checkout-api"} | json | event="http_request"
  | unwrap duration_ms [5m]
)
```

In CloudWatch Logs Insights, the same spirit with different syntax:

```
fields @timestamp, request_id, error_type
| filter level = "error" and service = "checkout-api"
| stats count(*) as errors by error_type, bin(5m)
| sort errors desc
```

And in Elasticsearch/OpenSearch, indexed JSON fields allow direct aggregations (terms on error_type, percentiles on duration_ms) from Kibana without writing DSL by hand. Three cross-cutting observations. One: all these queries depend on level, event, and the numeric fields existing with consistent types —the shared schema is what makes them possible. Two: the queries you use in incidents should be saved (saved queries, dashboards) because at 3 a.m. nobody remembers the unwrap syntax. Three: if a log query runs every five minutes to feed an alert, it should probably be a metric; alerts on logs are legitimate for specific patterns (a particular error_type, an audit event), not as a cheap substitute for the metrics system.

Migrating a legacy service without rewriting it

You rarely start from zero. The incremental path that works: first, switch the formatter to JSON without touching a single call site —in Python, ProcessorFormatter does exactly this: existing logging.getLogger() calls start coming

out as JSON with correct timestamp and level without modifying one line of application code. Second, add the context middleware: from that moment, even legacy logs carry `request_id`, which is 80% of the value. Third, convert the highest-value messages into events with fields ("`Order %s created for %s`" becomes `event="order_created"` with `order_id` and `user_id`), starting with the ones that appear in your recent incident searches —your platform's search history is literally the prioritized list of what to migrate. Fourth, add the canonical line, which requires no changes to existing code, only the middleware.

What does not work: the big-bang migration that renames every message at once (it breaks the dashboards and saved queries that depended on the old texts, all on the same day) and the "new standard" that only applies to new services (it guarantees two formats forever). Treat event names like an API with deprecation: if `order_created` must become `checkout_order_created`, emit both for a sprint and warn whoever has dashboards on top of it.

Hands-on: reconstructing a payment failure at 3:07 a.m.

Let's watch everything above working together. At 3:07 the alert fires: `checkout-api`'s 5xx error rate is over the threshold. You open the log platform and query the failed canonical lines from the last 15 minutes. In Loki:

```
{service="checkout-api"} | json
| event = "http_request" | status >= 500
| line_format "{{.request_id}} {{.path}} {{.error_type}}"
```

Result: 47 failed requests, all on `POST /orders`, all with `error_type=TimeoutError` and `payment_provider=stripe`. You know the what; you need the where. You take one specific `request_id` and ask for its full story, across every service:

```
{namespace="prod"} | json
| request_id = "3f2a91c0-7b2e-4d1a-9c55-8f0f2b6a1e77"
```

Ordered by timestamp, out come the `checkout-api` lines (`order_created`, `payment_started`), the `payments-svc` one (`provider_request_sent...` then nothing for 10 seconds), and the final timeout error. The same line carries `trace_id`: one click and you are looking at the trace in your tracing backend, where the external provider's span shows the exact 10,000 ms. The cause: a degraded payment provider. Confirmation comes from `payments-svc`'s circuit breaker warning events —which appear in the same search because every service shares the schema.

Total time: a few minutes, three queries, zero grep. Every piece of this guide played its part: the canonical line gave the pattern (grouping by `error_type` and `payment_provider`), the propagated `request_id` tied the services together, the `trace_id` jumped to the trace, the shared schema allowed a single cross-service query, and the timeouts logged with numeric fields made the distribution visible. Without structured logging, this same investigation is two hours of grep across four different formats.

The incident's echo also flows through the logs. For the postmortem, the complete sequence of events with exact timestamps is already written —the timeline is copied from the query, not reconstructed from memory. And the follow-up actions come from the same material: the provider's 10 s timeout was too generous (the `timeout_ms` field gave it away), an alert on the circuit breaker's warnings was missing, and the `provider_request_sent` event deserved a `provider_latency_ms` field that now gets added to the catalog. A well-logged incident leaves the system better instrumented than before: that is the virtuous spiral that justifies everything in this guide.

Quick FAQ

Should I migrate my legacy services all at once? No. Start at the edge (where the `request_id` is born) and with the services that show up most in incidents. The shipper's parsers can translate legacy formats in the meantime.

logfmt or JSON? logfmt (key=value) is easier on the eyes and sufficient for flat structures, but JSON wins as soon as you need nesting, types, and universal compatibility with shippers and platforms. When in doubt, JSON.

What about local development? A readable renderer (`ConsoleRenderer` in `structlog`, `pino-pretty` in Node) toggled by environment variable. JSON locally only punishes your eyes without adding anything.

Isn't structured logging slower? JSON serialization has a cost, but with pino (worker-thread transports) and structlog (cheap level filtering, cached loggers) it is irrelevant next to the cost of a single database query. The real cost of logging is ingestion and retention, and that is controlled with sampling, not by going back to plain text.

How many fields are too many? Stripe's canonical line carries dozens and it works. The practical limit is not the number but the discipline: scalar fields, stable names, pinned types. A well-named new field is cheap; renaming one used by dashboards is very expensive.

Should I also keep logs in a local file? In containers, no: stdout is the contract, and rotation is the platform's problem. Local files inside the container disappear with the pod, fill ephemeral disks, and nobody collects them. The exception is environments without an orchestrator (a classic VM), where a file with logrotate is still reasonable.

What about the load balancer or ingress access logs? Enable them and treat them as one more structured source (ALB and nginx can emit JSON). They are the only view of the requests that died before reaching your application —TLS timeouts, 502s from the proxy itself— and they complete the story your canonical lines cannot tell.

Where do I start if I have none of this today? In one afternoon: JSON in production (structlog or pino with this guide's configuration), a request_id middleware, and redaction of the three or four obviously sensitive fields. That foundation already transforms your next investigation. The following week: the canonical line and a documented minimal schema. Everything else —OTLP, fine-grained sampling, the event catalog— gets added when volume or team size demands it, on top of a foundation you will not have to redo.

Production checklist

- JSON output in production; human-readable renderer only in development.
- ISO 8601 UTC timestamp, level, event, and service on every log.
- request_id generated or propagated at the edge, present on every line of the request.
- trace_id and span_id injected if you use OpenTelemetry.
- Third-party library logs unified into the same pipeline (ProcessorFormatter in Python).
- Secret and PII redaction configured in the logger, not at each call site, with a test verifying it.
- A canonical line per request with status, duration, and business fields; job_started/job_finished events in workers.
- Context (request_id, trace context) propagated inside queue messages.
- A documented minimal shared schema, aligned with OTel semantic conventions.
- info level by default; debug toggleable via environment variable without redeploying.
- Health checks excluded; repetitive noise sampled; errors and canonical lines never sampled.
- Retention by level and an ingestion budget watched by a volume alert.
- Logs to stdout/stderr: rotation and shipping are the platform's job (Docker, Kubernetes, systemd), not your application's.

None of this is exotic: two mature libraries, a well-configured shipper, and a handful of agreed conventions. The payoff arrives the day a payment fails in cascade across three services and you reconstruct the whole story with a single search by request_id —and the log platform's bill holds no surprises, because the noise stayed at the edge.