Logging sucks.

loggingsucks.com

And here's how to make it better.

by Boris Tane

2024-12-20T03:14:22.847Z INFO HttpServer started successfully binding=0.0.0.0:3000 pid=28471 env=production version=2.4.1 node_env=production cluster_mode=enabled workers=4

2024-12-20T03:14:22.912Z debug PostgreSQL connection pool initialized host=db.internal:5432 database=main pool_size=20 ssl_mode=require idle_timeout=10000ms max_lifetime=1800000ms

2024-12-20T03:14:23.156Z INFO Incoming request method=GET path=/api/v1/users/me ip=192.168.1.42 user_agent="Mozilla/5.0" request_id=req_8f7a2b3c trace_id=abc123def456

2024-12-20T03:14:23.201Z debug JWT token validation started issuer=auth.company.com audience=api.company.com exp=1703044800 iat=1703041200 sub=user_abc123 scope="read write"

2024-12-20T03:14:23.445Z WARN Slow database query detected duration_ms=847 query="SELECT u.*, o.name FROM users u JOIN orgs o ON u.org_id = o.id WHERE u.org_id = $1 AND u.deleted_at IS NULL" rows_returned=2847

2024-12-20T03:14:23.892Z debug Redis cache lookup failed key=users:org_12345:list:v2 ttl_seconds=3600 fallback_strategy=database cache_cluster=redis-prod-01 latency_ms=2

2024-12-20T03:14:24.156Z info Request completed successfully status=200 duration_ms=1247 bytes_sent=48291 request_id=req_8f7a2b3c cache_hit=false db_queries=3 external_calls=1

2024-12-20T03:14:24.312Z ERROR Database connection pool exhausted active_connections=20 waiting_requests=147 timeout_ms=30000 service=postgres suggestion="Consider increasing pool_size or optimizing queries"

2024-12-20T03:14:24.445Z warn Retrying failed HTTP request attempt=1 max_attempts=3 backoff_ms=100 error_code=ETIMEDOUT target_service=payment-gateway endpoint=/v1/charges circuit_state=closed

2024-12-20T03:14:25.112Z INFO Circuit breaker state transition service=payment-api previous_state=closed current_state=open failure_count=5 failure_threshold=5 reset_timeout_ms=30000

2024-12-20T03:14:25.445Z debug Background job executed successfully job_id=job_9x8w7v6u type=weekly_email_digest duration_ms=2341 emails_sent=1847 failures=3 queue=default priority=low

2024-12-20T03:14:26.201Z ERROR Memory pressure critical heap_used_bytes=1932735283 heap_limit_bytes=2147483648 gc_pause_ms=847 gc_type=major rss_bytes=2415919104 external_bytes=8847291

2024-12-20T03:14:26.556Z WARN Rate limit threshold approaching user_id=user_abc123 current_requests=890 limit=1000 window_seconds=60 remaining=110 reset_at=2024-12-20T03:15:00Z

2024-12-20T03:14:27.112Z info WebSocket connection established client_id=ws_7f8g9h2j protocol=wss rooms=["team_updates","notifications","presence"] user_id=user_abc123 ip=192.168.1.42

2024-12-20T03:14:27.445Z debug Kafka message consumed successfully topic=user-events partition=3 offset=1847291 key=user_abc123 consumer_group=api-consumers lag=12 processing_time_ms=45

2024-12-20T03:14:28.112Z INFO Health check passed service=api-gateway uptime_seconds=847291 active_connections=142 memory_usage_percent=73 cpu_usage_percent=45 status=healthy version=2.4.1

2024-12-20T03:14:28.556Z debug S3 upload completed bucket=company-uploads key=avatars/user_abc123/profile.jpg size_bytes=245891 content_type=image/jpeg duration_ms=892 region=us-east-1

2024-12-20T03:14:29.112Z warn Deprecated API version detected endpoint=/api/v1/legacy/users version=v1 recommended_version=v3 deprecation_date=2025-01-15 client_id=mobile-app-ios

2024-12-20T03:14:22.847Z INFO HttpServer started successfully binding=0.0.0.0:3000 pid=28471 env=production version=2.4.1 node_env=production cluster_mode=enabled workers=4

2024-12-20T03:14:22.912Z debug PostgreSQL connection pool initialized host=db.internal:5432 database=main pool_size=20 ssl_mode=require idle_timeout=10000ms max_lifetime=1800000ms

2024-12-20T03:14:23.156Z INFO Incoming request method=GET path=/api/v1/users/me ip=192.168.1.42 user_agent="Mozilla/5.0" request_id=req_8f7a2b3c trace_id=abc123def456

2024-12-20T03:14:23.201Z debug JWT token validation started issuer=auth.company.com audience=api.company.com exp=1703044800 iat=1703041200 sub=user_abc123 scope="read write"

2024-12-20T03:14:23.445Z WARN Slow database query detected duration_ms=847 query="SELECT u.*, o.name FROM users u JOIN orgs o ON u.org_id = o.id WHERE u.org_id = $1 AND u.deleted_at IS NULL" rows_returned=2847

2024-12-20T03:14:23.892Z debug Redis cache lookup failed key=users:org_12345:list:v2 ttl_seconds=3600 fallback_strategy=database cache_cluster=redis-prod-01 latency_ms=2

2024-12-20T03:14:24.156Z info Request completed successfully status=200 duration_ms=1247 bytes_sent=48291 request_id=req_8f7a2b3c cache_hit=false db_queries=3 external_calls=1

2024-12-20T03:14:24.312Z ERROR Database connection pool exhausted active_connections=20 waiting_requests=147 timeout_ms=30000 service=postgres suggestion="Consider increasing pool_size or optimizing queries"

2024-12-20T03:14:24.445Z warn Retrying failed HTTP request attempt=1 max_attempts=3 backoff_ms=100 error_code=ETIMEDOUT target_service=payment-gateway endpoint=/v1/charges circuit_state=closed

2024-12-20T03:14:25.112Z INFO Circuit breaker state transition service=payment-api previous_state=closed current_state=open failure_count=5 failure_threshold=5 reset_timeout_ms=30000

2024-12-20T03:14:25.445Z debug Background job executed successfully job_id=job_9x8w7v6u type=weekly_email_digest duration_ms=2341 emails_sent=1847 failures=3 queue=default priority=low

2024-12-20T03:14:26.201Z ERROR Memory pressure critical heap_used_bytes=1932735283 heap_limit_bytes=2147483648 gc_pause_ms=847 gc_type=major rss_bytes=2415919104 external_bytes=8847291

2024-12-20T03:14:26.556Z WARN Rate limit threshold approaching user_id=user_abc123 current_requests=890 limit=1000 window_seconds=60 remaining=110 reset_at=2024-12-20T03:15:00Z

2024-12-20T03:14:27.112Z info WebSocket connection established client_id=ws_7f8g9h2j protocol=wss rooms=["team_updates","notifications","presence"] user_id=user_abc123 ip=192.168.1.42

2024-12-20T03:14:27.445Z debug Kafka message consumed successfully topic=user-events partition=3 offset=1847291 key=user_abc123 consumer_group=api-consumers lag=12 processing_time_ms=45

2024-12-20T03:14:28.112Z INFO Health check passed service=api-gateway uptime_seconds=847291 active_connections=142 memory_usage_percent=73 cpu_usage_percent=45 status=healthy version=2.4.1

2024-12-20T03:14:28.556Z debug S3 upload completed bucket=company-uploads key=avatars/user_abc123/profile.jpg size_bytes=245891 content_type=image/jpeg duration_ms=892 region=us-east-1

2024-12-20T03:14:29.112Z warn Deprecated API version detected endpoint=/api/v1/legacy/users version=v1 recommended_version=v3 deprecation_date=2025-01-15 client_id=mobile-app-ios
Your logs are lying to you. Not maliciously. They're just not equipped to tell the truth.

You've probably spent hours grep-ing through logs trying to understand why a user couldn't check out, why that webhook failed, or why your p99 latency spiked at 3am. You found nothing useful. Just timestamps and vague messages that mock you with their uselessness.

This isn't your fault. **Logging, as it's commonly practiced, is fundamentally broken.** And no, slapping OpenTelemetry on your codebase won't magically fix it.

Let me show you what's wrong, and more importantly, how to fix it.

The Core Problem

Logs were designed for a different era. An era of monoliths, single servers, and problems you could reproduce locally. Today, a single user request might touch 15 services, 3 databases, 2 caches, and a message queue. Your logs are still acting like it's 2005.

Here's what a typical logging setup looks like:

The Log Chaos Simulator

Watch logs stream in from a single checkout request. Then crank up concurrent users to see the chaos unfold.

0 log lines per second from 0 concurrent users

Click "Process Order" to see logs appear

That's 17 log lines for a single successful request. Now multiply that by 10,000 concurrent users. You've got 130,000 log lines per second. Most of them saying absolutely nothing useful.

But here's the real problem: when something goes wrong, these logs won't help you. They're missing the one thing you need: **context**.

Why String Search is Broken

When a user reports "I can't complete my purchase," your first instinct is to search your logs. You type their email, or maybe their user ID, and hit enter.

The Futile Search

Search for "user-123" and find the log line, but notice how much context you're missing. The error happened, but why?

1 [INFO] 2025-01-15 10:23:41 Server started on port 3000

2 [DEBUG] 2025-01-15 10:23:42 Database connection pool initialized

3 [INFO] 2025-01-15 10:23:45 Processing request for user user-123

4 [DEBUG] 2025-01-15 10:23:45 Cache lookup: session_abc123

5 [INFO] 2025-01-15 10:23:46 Request received from 192.168.1.50

6 [DEBUG] 2025-01-15 10:23:46 Parsing JSON body, size: 1.2KB

7 [INFO] 2025-01-15 10:23:47 user_id=user-123 action=checkout started

8 [DEBUG] 2025-01-15 10:23:47 Loading cart items from database

9 [INFO] 2025-01-15 10:23:48 Request from user-1234 processed

10 [DEBUG] 2025-01-15 10:23:48 Validating payment method

11 [WARN] 2025-01-15 10:23:49 Slow query detected: 450ms

12 [INFO] 2025-01-15 10:23:49 {"user":"user-123","event":"payment_failed"}

13 [DEBUG] 2025-01-15 10:23:50 Retrying payment, attempt 2

14 [ERROR] 2025-01-15 10:23:50 Payment gateway timeout

15 [INFO] 2025-01-15 10:23:51 UserID: user-123 - Session timeout warning

16 [DEBUG] 2025-01-15 10:23:51 Refreshing session token

17 [INFO] 2025-01-15 10:23:52 Request completed for user-12345

18 [DEBUG] 2025-01-15 10:23:52 Cleaning up temporary files

19 [INFO] 2025-01-15 10:23:53 [user-123] Cart updated, 3 items

20 [DEBUG] 2025-01-15 10:23:53 Broadcasting cart update event

21 [INFO] 2025-01-15 10:23:54 Health check passed

22 [DEBUG] 2025-01-15 10:23:54 Memory usage: 45%

23 [INFO] 2025-01-15 10:23:55 Background job started: email-queue

24 [DEBUG] 2025-01-15 10:23:55 Processing 12 pending emails

25 [WARN] 2025-01-15 10:23:56 Rate limit approaching for IP 192.168.1.50

26 [INFO] 2025-01-15 10:23:56 WebSocket connection established

27 [DEBUG] 2025-01-15 10:23:57 Subscribing to channels: [orders, notifications]

28 [INFO] 2025-01-15 10:23:57 Order #4521 created successfully

29 [DEBUG] 2025-01-15 10:23:58 Sending order confirmation email

30 [INFO] 2025-01-15 10:23:58 Inventory updated for SKU-789
Results

Enter a search term to find matching logs
String search treats logs as bags of characters. It has no understanding of structure, no concept of relationships, no way to correlate events across services.

> When you search for "user-123", you might find it logged 47 different ways across your codebase:

- `user-123`
- `user_id=user-123`
- `{"userId": "user-123"}`
- `[USER:user-123]`
- `processing user: user-123`

And those are just the logs that *include* the user ID. What about the downstream service that only logged the order ID? Now you need a second search. And a third. You're playing detective with one hand tied behind your back.

> The fundamental problem: logs are optimized for *writing*, not for *querying*.

Developers write `console.log("Payment failed")` because it's easy in the moment. Nobody thinks about the poor soul who'll be searching for this at 2am during an outage.

Let's Define Some Terms

Before I show you the fix, let me define some terms. These get thrown around a lot, often incorrectly.

Structured Logging: Logs emitted as key-value pairs (usually JSON) instead of plain strings. `{"event": "payment_failed", "user_id": "123"}` instead of `"Payment failed for user 123"`. Structured logging is necessary but not sufficient.

Cardinality: The number of unique values a field can have. `user_id` has high cardinality (millions of unique values). `http_method` has low cardinality (GET, POST, PUT, DELETE, etc.). High cardinality fields are what make logs actually useful for debugging.

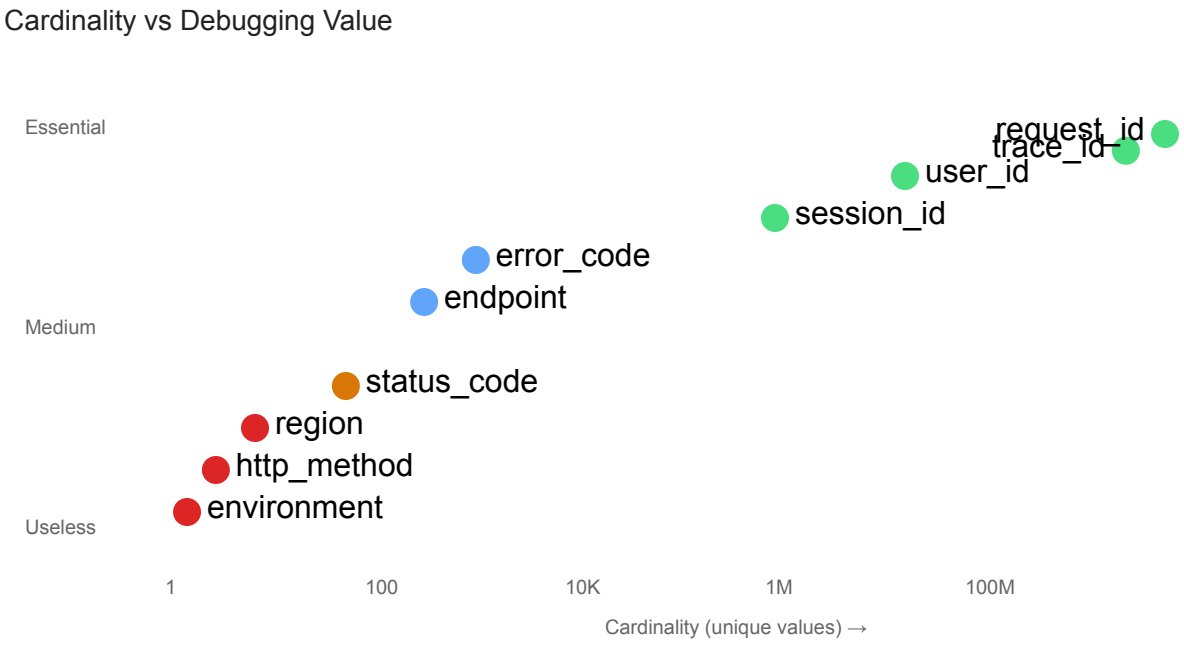
Dimensionality: The number of fields in your log event. A log with 5 fields has low dimensionality. A log with 50 fields has high dimensionality. More dimensions = more questions you can answer.

Wide Event: A single, context-rich log event emitted per request per service. Instead of 13 log lines for one request, you emit 1 line with 50+ fields containing everything you might need to debug.

Canonical Log Line: Another term for wide event, popularized by Stripe. One log line per request that serves as the authoritative record of what happened.

Cardinality Explorer

High-cardinality fields (like user_id) are the most valuable for debugging. Hover over points to see why each field matters.



The irony: Most logging systems charge by volume and choke on high-cardinality fields. This is backwards. High cardinality is exactly what you need for debugging.

OpenTelemetry Won't Save You

I see this take constantly: "Just use OpenTelemetry and your observability problems are solved."

No. OpenTelemetry is a **protocol and a set of SDKs**. It standardizes how telemetry data (logs, traces, metrics) is collected and exported. This is genuinely useful: it means you're not locked into a specific vendor's format.

But here's what OpenTelemetry does NOT do:

1. **It doesn't decide what to log.** You still have to instrument your code deliberately.
2. **It doesn't add business context.** If you don't add the user's subscription tier, their cart value, or the feature flags enabled, OTel won't magically know.
3. **It doesn't fix your mental model.** If you're still thinking in terms of "log statements," you'll just emit bad telemetry in a standardized format.

The OTel Reality Check

OpenTelemetry auto-instrumentation captures basic spans. But the context you add makes all the difference.

Lazy instrumentation

```
// "We use OpenTelemetry!"
import { trace } from '@opentelemetry/api';

app.post('/checkout', async (req, res) => {
  const span = trace.getActiveSpan();

  try {
    const order = await processOrder(req.body);
    res.json(order);
  } catch (error) {
    span?.recordException(error);
    res.status(500).json({ error: 'Failed' });
  }
});
```

Deliberate instrumentation

```
// Actually useful instrumentation
import { trace } from '@opentelemetry/api';

app.post('/checkout', async (req, res) => {
  const span = trace.getActiveSpan();
  const user = req.user;

  // Add business context BEFORE processing
  span?.setAttributes({
    'user.id': user.id,
    'user.subscription': user.plan,
    'user.lifetime_value': user.ltv,
    'cart.item_count': req.body.items.length,
    'cart.total_cents': req.body.total,
    'feature_flags': JSON.stringify(user.flags),
    'checkout.payment_method': req.body.paymentMethod,
  });

  try {
    const order = await processOrder(req.body);

    span?.setAttributes({
      'order.id': order.id,
      'order.status': order.status,
      'payment.provider': order.paymentProvider,
      'payment.latency_ms': order.paymentLatency,
    });

    res.json(order);
  } catch (error) {
    span?.setAttributes({
      'error.type': error.name,
      'error.code': error.code,
      'error.retriable': error.retriable,
    });
    span?.recordException(error);
    res.status(500).json({ error: 'Failed' });
  }
});
```

Same library. Same protocol. Wildly different debugging experience.

OTel is just plumbing — YOU decide what flows through it.

OpenTelemetry is a delivery mechanism. It doesn't know that `user-789` is a premium customer who's been with you for 3 years and just tried to spend $160. **You** have to tell it.

The Fix: Wide Events / Canonical Log Lines

Here's the mental model shift that changes everything:

> Instead of logging *what your code is doing*, log *what happened to this request*.

Stop thinking about logs as a debugging diary. Start thinking about them as a structured record of business events.

For each request, emit **one wide event** per service hop. This event should contain every piece of context that might be useful for debugging. Not just what went wrong, but the full picture of the request.

Build a Wide Event

Toggle fields on/off to build your wide event. Watch the JSON update and see which debugging scenarios each field enables.

16 fields — moderate

```
{
  "request_id": "req_8bf7ec2d",
  "trace_id": "abc123def456",
  "method": "POST",
  "path": "/api/checkout",
  "status_code": 500,
  "duration_ms": 1247,
  "timestamp": "2025-01-15T10:23:45.612Z",
  "user_id": "user_456",
  "session_id": "sess_abc123",
  "subscription_tier": "premium",
  "service_name": "checkout-service",
  "service_version": "2.4.1",
  "region": "us-east-1",
  "error_type": "PaymentError",
  "error_code": "card_declined",
  "error_message": "Card declined by issuer"
}
```

Can you answer these questions with your event?

No

Why did user X's checkout fail?

Missing: payment_method

Yes

Are premium users experiencing more errors?

No

Which deployment caused the latency regression?

Missing: deployment_id

No

What's the error rate for the new checkout feature?

Missing: feature_flags
Here's what a wide event looks like in practice:

```
{
  "timestamp": "2025-01-15T10:23:45.612Z",
  "request_id": "req_8bf7ec2d",
  "trace_id": "abc123",

  "service": "checkout-service",
  "version": "2.4.1",
  "deployment_id": "deploy_789",
  "region": "us-east-1",

  "method": "POST",
  "path": "/api/checkout",
  "status_code": 500,
  "duration_ms": 1247,

  "user": {
    "id": "user_456",
    "subscription": "premium",
    "account_age_days": 847,
    "lifetime_value_cents": 284700
  },

  "cart": {
    "id": "cart_xyz",
    "item_count": 3,
    "total_cents": 15999,
    "coupon_applied": "SAVE20"
  },

  "payment": {
    "method": "card",
    "provider": "stripe",
    "latency_ms": 1089,
    "attempt": 3
  },

  "error": {
    "type": "PaymentError",
    "code": "card_declined",
    "message": "Card declined by issuer",
    "retriable": false,
    "stripe_decline_code": "insufficient_funds"
  },

  "feature_flags": {
    "new_checkout_flow": true,
    "express_payment": false
  }
}
```

One event. Everything you need. When this user complains, you search for `user_id = "user_456"` and you instantly know:

- They're a premium customer (high priority)
- They've been with you for over 2 years (very high priority)
- The payment failed on the 3rd attempt
- The actual reason: insufficient funds
- They were using the new checkout flow (potential correlation?)

No grep-ing. No guessing. No second search.

The Queries You Can Now Run

With wide events, you're not searching text anymore. You're querying structured data. The difference is night and day.

Query Playground

Click a preset query or write your own SQL. With wide events, complex questions become simple queries.

Select a query above or write your own

This is the superpower of wide events combined with high-cardinality, high-dimensionality data. You're not searching logs anymore. You're running analytics on your production traffic.

Implementing Wide Events

Here's a practical implementation pattern. The key insight: build the event throughout the request lifecycle, then emit once at the end.

```
// middleware/wideEvent.ts
export function wideEventMiddleware() {
  return async (ctx, next) => {
    const startTime = Date.now();

    // Initialize the wide event with request context
    const event: Record<string, unknown> = {
      request_id: ctx.get('requestId'),
      timestamp: new Date().toISOString(),
      method: ctx.req.method,
      path: ctx.req.path,
      service: process.env.SERVICE_NAME,
      version: process.env.SERVICE_VERSION,
      deployment_id: process.env.DEPLOYMENT_ID,
      region: process.env.REGION,
    };

    // Make the event accessible to handlers
    ctx.set('wideEvent', event);

    try {
      await next();
      event.status_code = ctx.res.status;
      event.outcome = 'success';
    } catch (error) {
      event.status_code = 500;
      event.outcome = 'error';
      event.error = {
        type: error.name,
        message: error.message,
        code: error.code,
        retriable: error.retriable ?? false,
      };
      throw error;
    } finally {
      event.duration_ms = Date.now() - startTime;

      // Emit the wide event
      logger.info(event);
    }
  };
}
```

Then in your handlers, you enrich the event with business context:

```
app.post('/checkout', async (ctx) => {
  const event = ctx.get('wideEvent');
  const user = ctx.get('user');

  // Add user context
  event.user = {
    id: user.id,
    subscription: user.subscription,
    account_age_days: daysSince(user.createdAt),
    lifetime_value_cents: user.ltv,
  };

  // Add business context as you process
  const cart = await getCart(user.id);
  event.cart = {
    id: cart.id,
    item_count: cart.items.length,
    total_cents: cart.total,
    coupon_applied: cart.coupon?.code,
  };

  // Process payment
  const paymentStart = Date.now();
  const payment = await processPayment(cart, user);

  event.payment = {
    method: payment.method,
    provider: payment.provider,
    latency_ms: Date.now() - paymentStart,
    attempt: payment.attemptNumber,
  };

  // If payment fails, add error details
  if (payment.error) {
    event.error = {
      type: 'PaymentError',
      code: payment.error.code,
      stripe_decline_code: payment.error.declineCode,
    };
  }

  return ctx.json({ orderId: payment.orderId });
});
```

Wide Event Builder Simulator

Step through a request lifecycle and watch the wide event accumulate context. One event captures everything.

Step 1 of 6

Request Received

Initialize event with request context

```
const event = {
  request_id: ctx.get('requestId'),
  timestamp: new Date().toISOString(),
  method: ctx.req.method,
  path: ctx.req.path,
  service: 'checkout-service',
};
```

Live Event

5 fields

```
{
  "request_id": "req_8bf7ec2d",
  "timestamp": "2025-01-15T10:23:45.612Z",
  "method": "POST",
  "path": "/api/checkout",
  "service": "checkout-service"
}
```

Sampling: Keeping Costs Under Control

"But Boris," I hear you saying, "if I log 50 fields per request at 10,000 requests per second, my observability bill will bankrupt me."

Valid concern. This is where **sampling** comes in.

Sampling means keeping only a percentage of your events. Instead of storing 100% of traffic, you might store 10% or 1%. At scale, this is the only way to stay sane (and solvent).

But naive sampling is dangerous. If you randomly sample 1% of traffic, you might accidentally drop the one request that explains your outage.

The Sampling Trap

Random sampling drops events blindly. Tail-based sampling keeps all errors and slow requests while sampling success.

Event Stream (10,000 events)
Success (85%) Slow (10%) Error (5%)

Sample Rate 10%

Events Kept	Success	Slow	Errors
89	75	12	2

At 10% random sampling, you have a **90% chance** of missing any specific error.

Tail Sampling

Tail sampling means you make the sampling decision *after* the request completes, based on its outcome.

The rules are simple:
1. **Always keep errors.** 100% of 500s, exceptions, and failures get stored.
2. **Always keep slow requests.** Anything above your p99 latency threshold.
3. **Always keep specific users.** VIP customers, internal testing accounts, flagged sessions.
4. **Randomly sample the rest.** Happy, fast requests? Keep 1-5%.

This gives you the best of both worlds: manageable costs, but you never lose the events that matter.

```
// Tail sampling decision function
function shouldSample(event: WideEvent): boolean {
  // Always keep errors
  if (event.status_code >= 500) return true;
  if (event.error) return true;

  // Always keep slow requests (above p99)
  if (event.duration_ms > 2000) return true;

  // Always keep VIP users
  if (event.user?.subscription === 'enterprise') return true;

  // Always keep requests with specific feature flags (debugging rollouts)
  if (event.feature_flags?.new_checkout_flow) return true;

  // Random sample the rest at 5%
  return Math.random() < 0.05;
}
```

Misconceptions

"Structured logging is the same as wide events"

No. Structured logging means your logs are JSON instead of strings. That's table stakes. Wide events are a *philosophy*: one comprehensive event per request, with all context attached. You can have structured logs that are still useless (5 fields, no user context, scattered across 20 log lines).

"We already use OpenTelemetry, so we're good"

You're using a delivery mechanism. OpenTelemetry doesn't decide what to capture. You do. Most OTel implementations I've seen capture the bare minimum: span name, duration, status. That's not enough. You need to deliberately instrument with business context.

"This is just tracing with extra steps"

Tracing gives you request flow across services (which service called which). Wide events give you context *within* a service. They're complementary. **Ideally, your wide events ARE your trace spans, enriched with all the context you need**.

"Logs are for debugging, metrics are for dashboards"

This distinction is artificial and harmful. Wide events can power both. Query them for debugging. Aggregate them for dashboards. The data is the same, just different views.

"High-cardinality data is expensive and slow"

It's expensive on *legacy logging systems* built for low-cardinality string search. Modern columnar databases (ClickHouse, BigQuery, etc.) are specifically designed for high-cardinality, high-dimensionality data. The tooling has caught up. Your practices should too.

The Payoff

When you implement wide events properly, debugging transforms from archaeology to analytics.

Instead of: *"The user said checkout failed. Let me grep through 50 services and hope I find something."*

You get: *"Show me all checkout failures for premium users in the last hour where the new checkout flow was enabled, grouped by error code."*

One query. Sub-second results. Root cause identified.

Your logs stop lying to you. They start telling the truth. The whole truth.