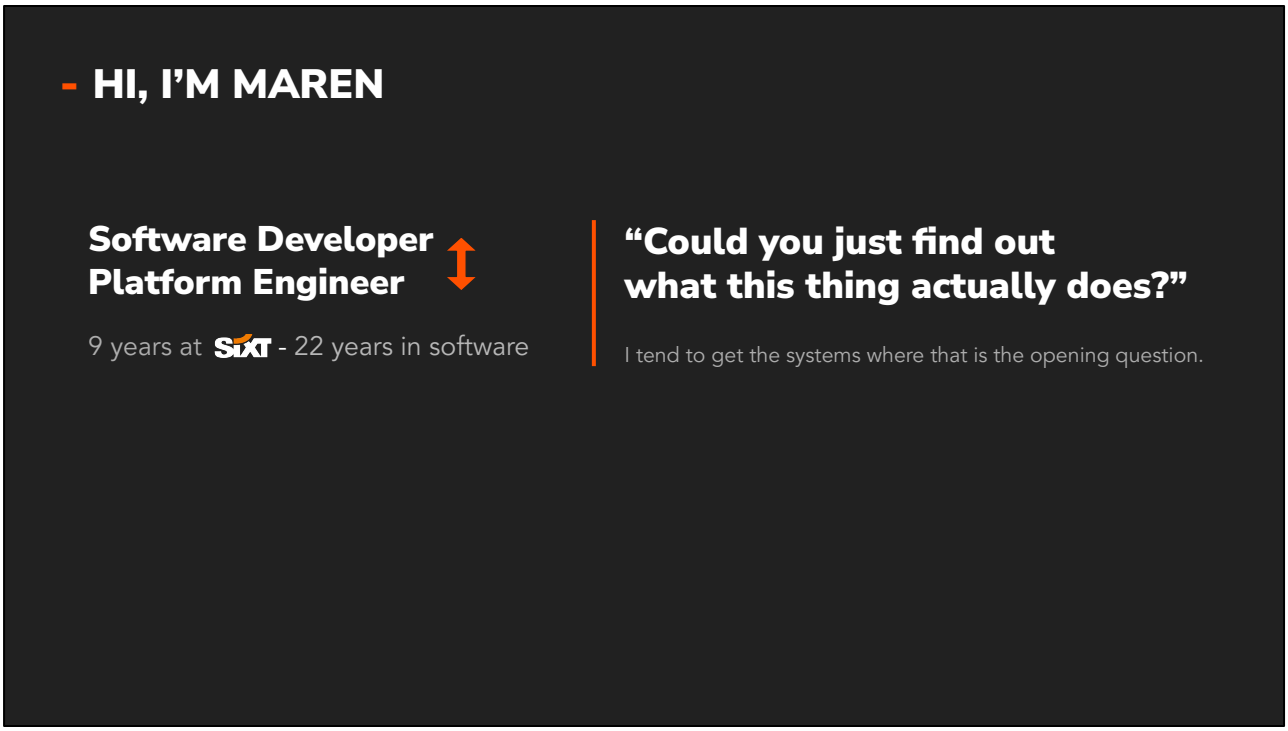
OPERATING THE UNOBSERVABLE

Building Observability for a System nobody wants to touch

Maren Afflerbach | SRECon Dublin

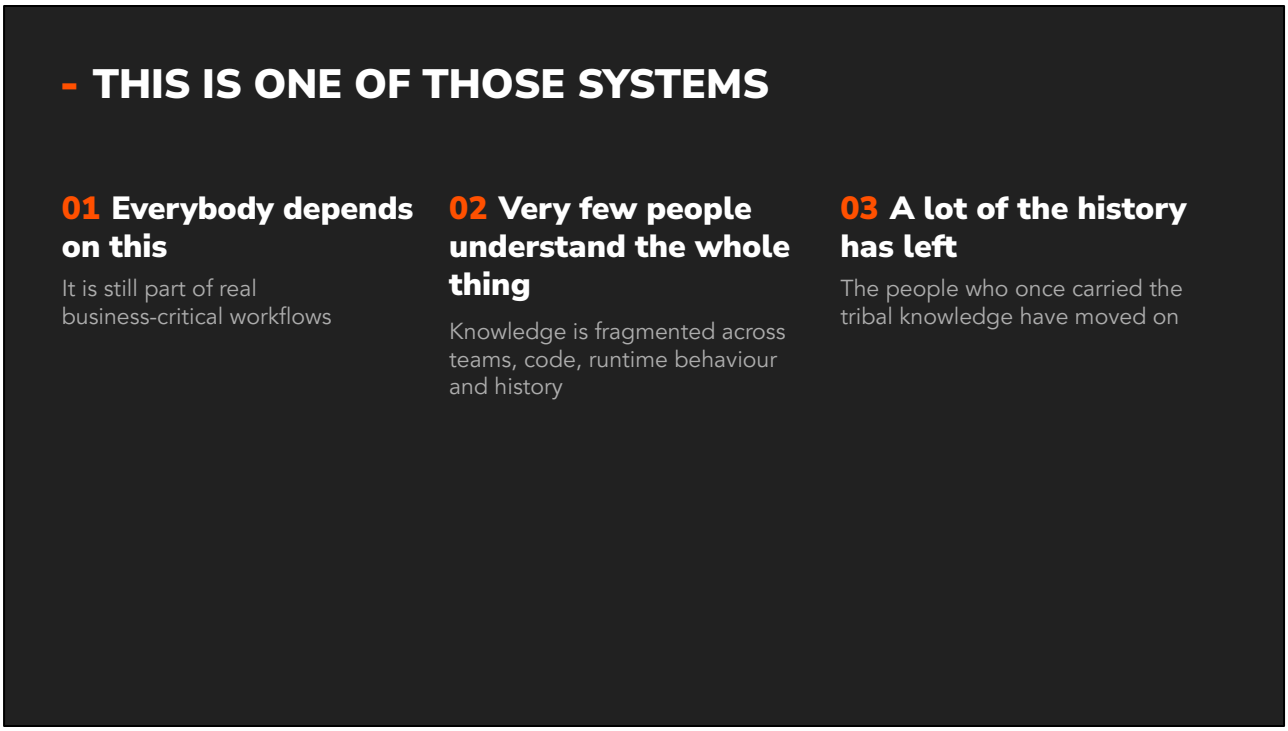




Hi, I'm Maren. I'm somewhere between a software developer and a platform engineer. I've been at SIXT for nine years,

I tend to get the systems where somebody eventually asks:

"Could you just find out what this thing actually does?"

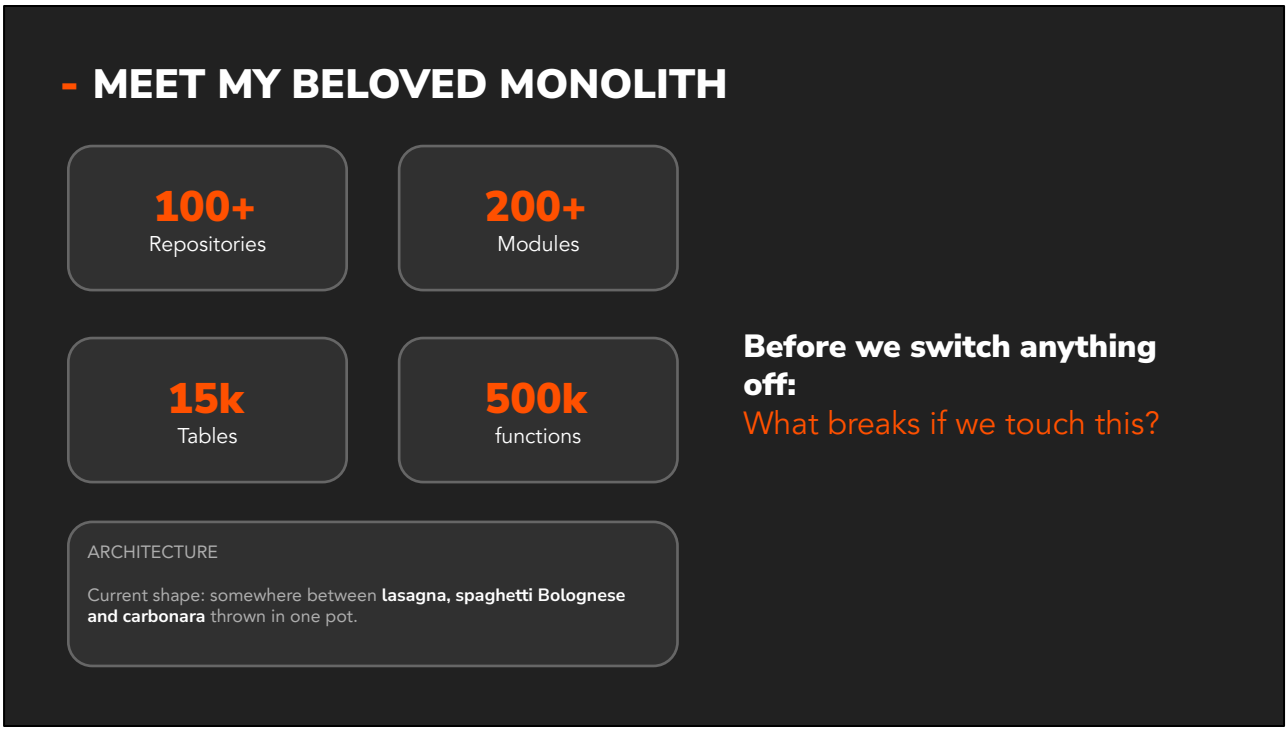


This is one of those systems.

Everybody depends on it, but very few people understand the whole thing anymore. Not every developer needs to work on the old stack, and many of the people who once carried the history and tribal knowledge have moved on.

So the first problem is not that the code is old.

The first problem is that the system still matters while our understanding of it has become incomplete.



Let me introduce you to my beloved monolith.

It spans more than 100 repositories and more than 200 modules. Our current analysis sees more than 15,000 source tables and more than half a million functions. Its main languages is PHP and COBOL.

You do not need to remember those numbers.

The point is that no person can reconstruct the whole system by reading it from one end to the other.

An architecture diagram tells us what somebody believed.

And I believe the architecture is currently somewhere between lasagna, spaghetti Bolognese, and carbonara thrown into one big pot.

The Flying Spaghetti Monster would probably be proud of us.

The language is not the hardest part. You can learn COBOL syntax.

The hard part is recovering the business and operational context that accumulated around it and over the decades.

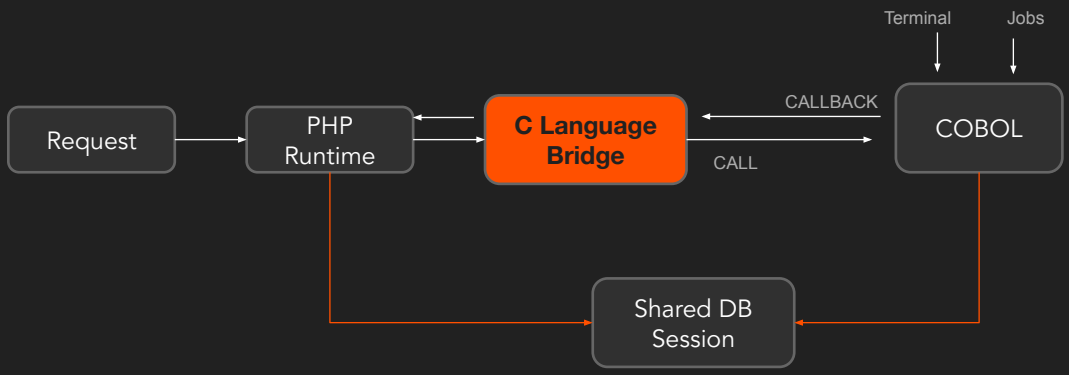
We want to decompose this monolith and eventually switch parts of it off, but Before changing a method whose last meaningful edit was may be older than some of its maintainers, we need to answer a reliability question:

What breaks if we touch this?

Source code tells us what *could* happen.

But it does not tell us what *still does* happen in production.

- ONE REQUEST. ONE TRANSACTION.



25+ years old. Still running critical business transactions

Here is what a single transaction looks like.

PHP and COBOL cooperate on critical business transactions.

A request can begin in PHP,

cross a language bridge through C bindings into COBOL,

execute SQL through a shared Oracle session, return to PHP, and commit as one unit.

COBOL can also call back into PHP.

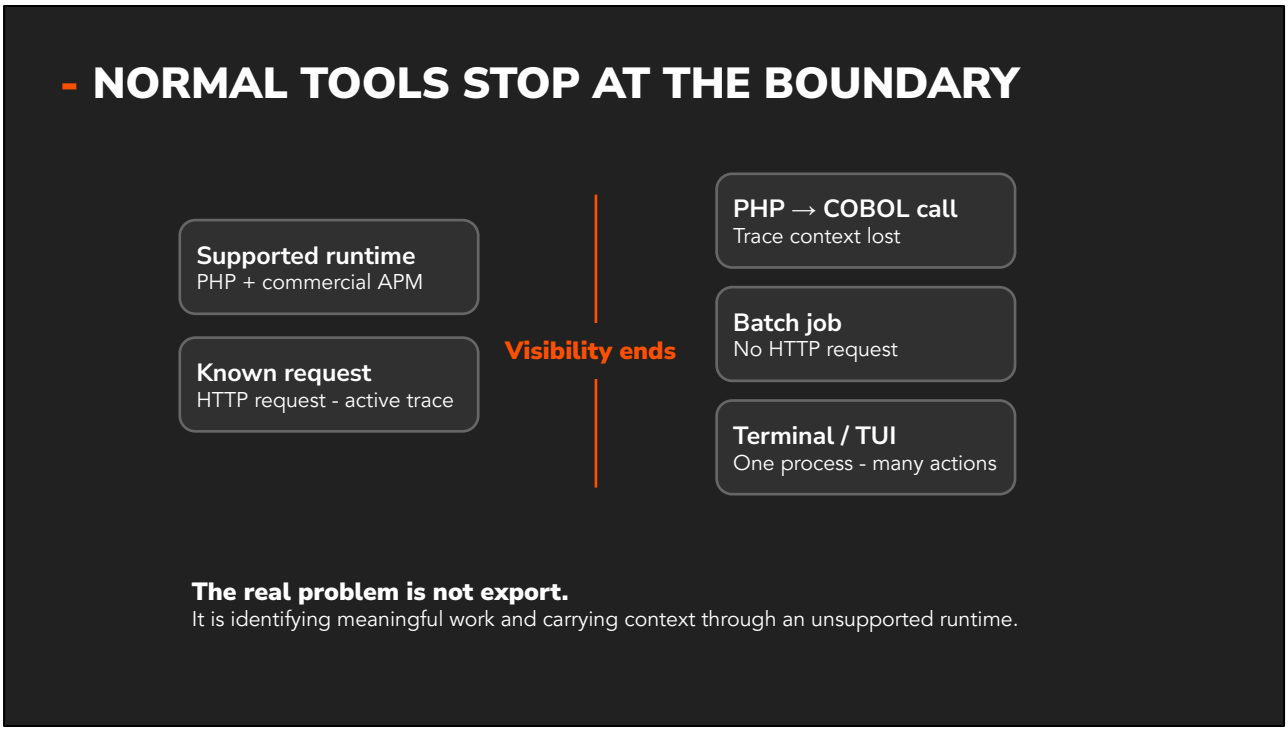
Other work enters through scheduled jobs and a terminal interface.

Parts of this system are more than 25 years old. Some parts are older than 40 years.

This is not a museum piece. It handles critical business transactions today.

We operate a fleet of hundreds of thousands of vehicles worldwide. In 2025, SIXT generated €4.28 billion in revenue and €400 million in earnings before taxes.

So yes: actual money is passing through this thing.



Standard observability works reasonably well when you're dealing with supported runtimes.

You install an agent, enable instrumentation, and suddenly you have traces. Wonderful.

And we actually had that. Our commercial APM gave us visibility into parts of our PHP applications.

But then we reached the boundary.

PHP would call COBOL. The transaction continued, but our trace didn't.

And that wasn't our only problem.

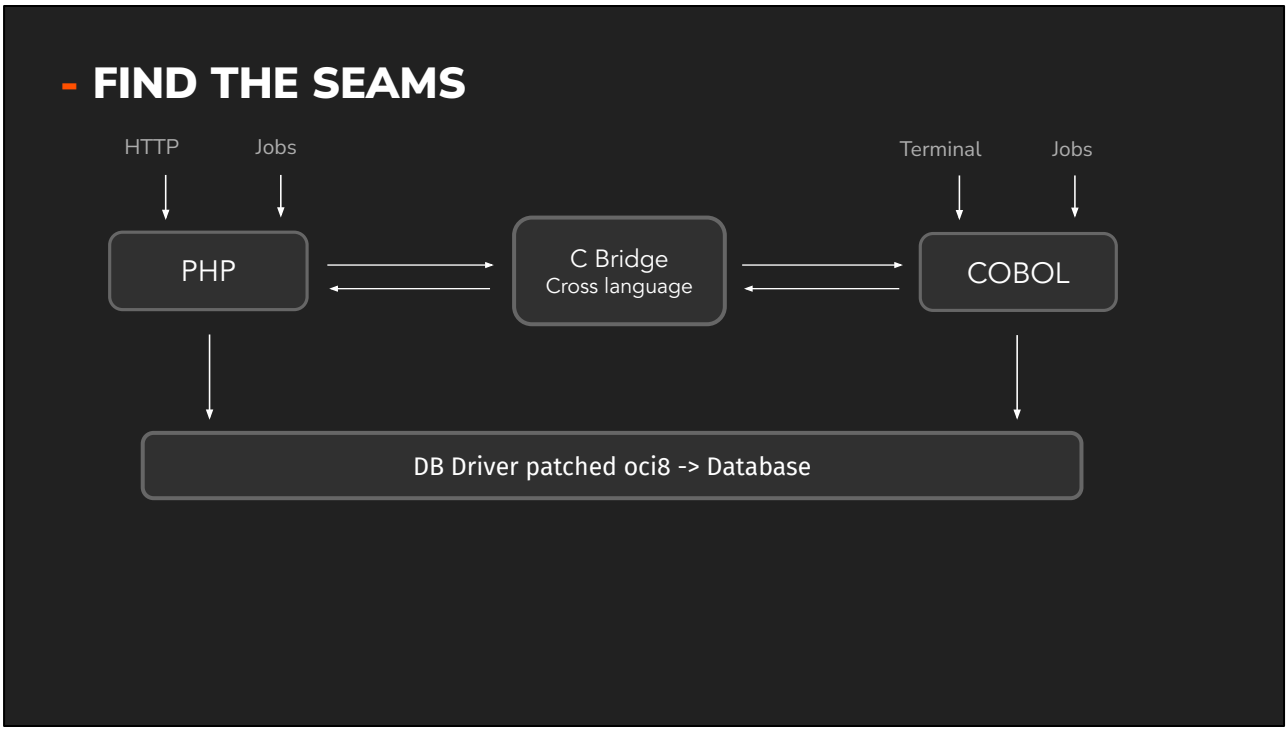
We had batch jobs that didn't start with an HTTP request.

And terminal applications where one process could run for hours, handling hundreds of completely unrelated business actions.

So even if we could instrument the process, what exactly should count as a

Exporting telemetry wasn't our biggest challenge.

First, we had to figure out what meaningful work actually looked like.



At this point, the question "Which agent can I install?" has a pretty short answer.

None. There is no agent.

But Instead of asking which runtime we could instrument, we asked:

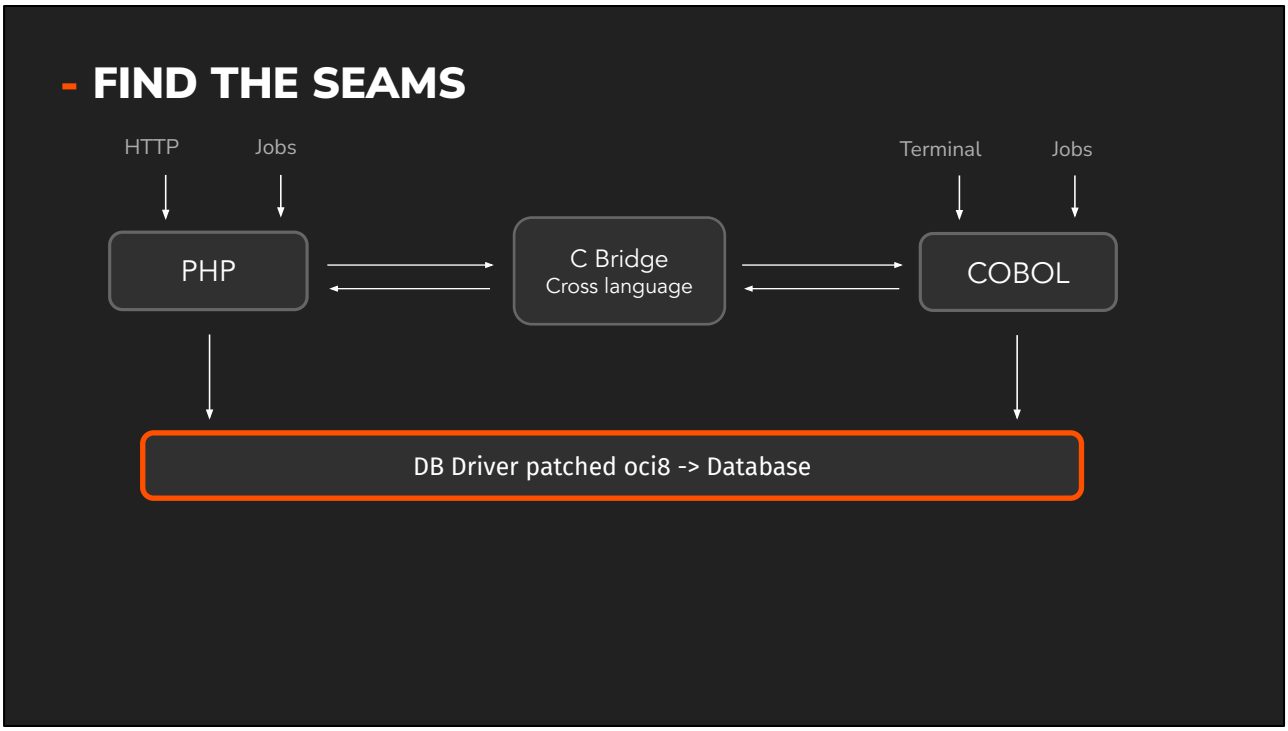
Which boundaries does the actual work have to pass through?

We looked at process entry points, scheduled jobs, terminal interactions,

the language bridge,

And database access.

And for each, we asked: What can we observe here? What do we miss? Can we preserve context? And perhaps most importantly, what happens if our instrumentation breaks?



Our first useful seam was the **database driver**.

Why the driver and not Oracle itself?

Oracle can tell us which statements executed. But it doesn't automatically tell us which PHP entry point or COBOL program was responsible.

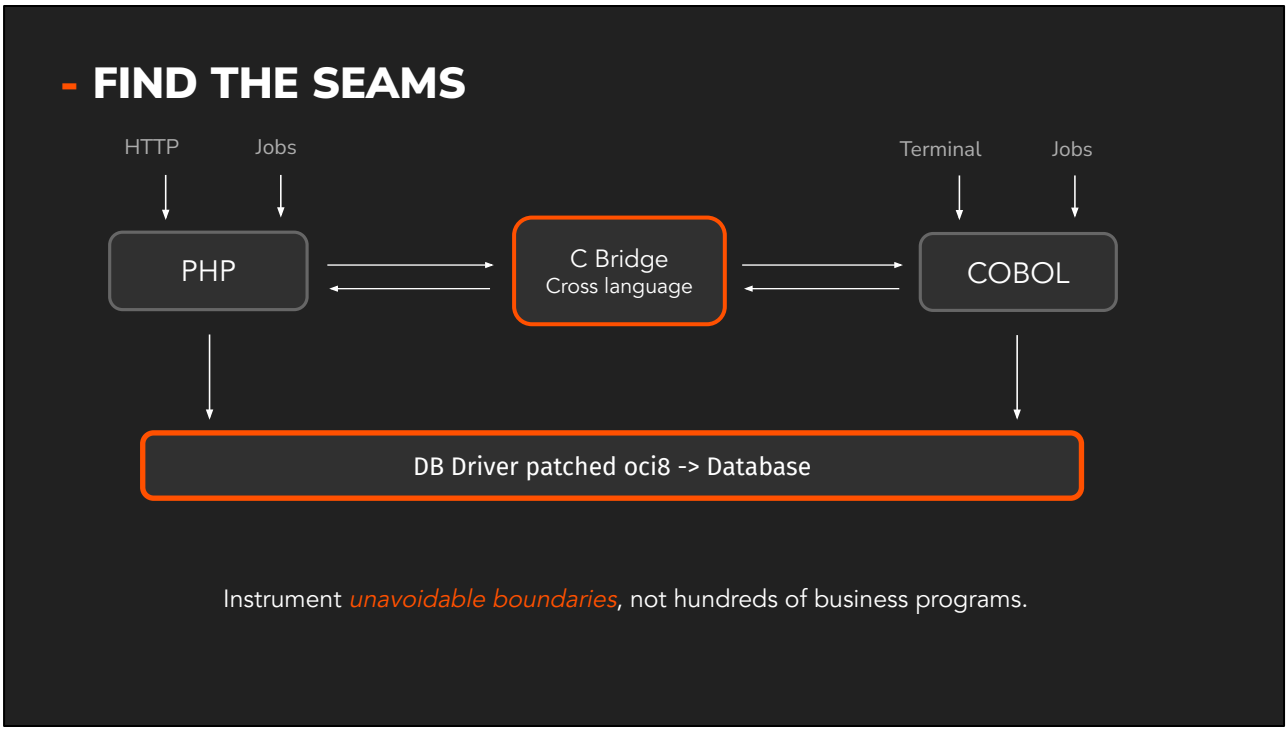
And here's the interesting part: And I want to point out **PHP and COBOL literally share the same Oracle session.**

We can open a transaction in PHP, execute SQL in COBOL, and commit it back in PHP. Or the other way around.

So looking at the database session alone doesn't tell us which language or which piece of application code actually executed a statement.

But at the driver boundary, we could capture information about the calling code.

And while we didn't have a common database abstraction in our application code, every Oracle statement passed through driver code we controlled.



Our second seam was the **generated C bridge** between PHP and COBOL.

Those two instrumentation points gave us broad coverage without having to modify hundreds of individual business programs.

We started looking for boundaries our application couldn't avoid.

- A QUERY BECOMES EVIDENCE

```
SELECT * FROM rental
WHERE customer_id = ?
AND created_at > ?       )-01'
```

Stable hash

Correlation Id Operation Type

Call Stack Program and Cursor names

Let’s go in to some details.

At the driver, we captured executed SQL and converted each statement into a stable fingerprint.

Literal strings and numbers were replaced for the hash, and keywords were normalised, so statements with different customer numbers or dates could be recognised as the same shape.

The fingerprint alone was not enough.

The driver sees every statement, but not the business intent behind it. The bridge sees every language crossing, but not the SQL. Each seam alone tells half the story.

That is why we joined them.

The fingerprint tells us that the same query shape appeared, but not why.

So we added Correlation Ids,

operation type, like Modifiyng, reading and type like insert, update, select etc.

On the PHP path, we captured the PHP backtrace before crossing into COBOL.

On the COBOL path, we retained the active program stack and used cursors.

That moved us from:

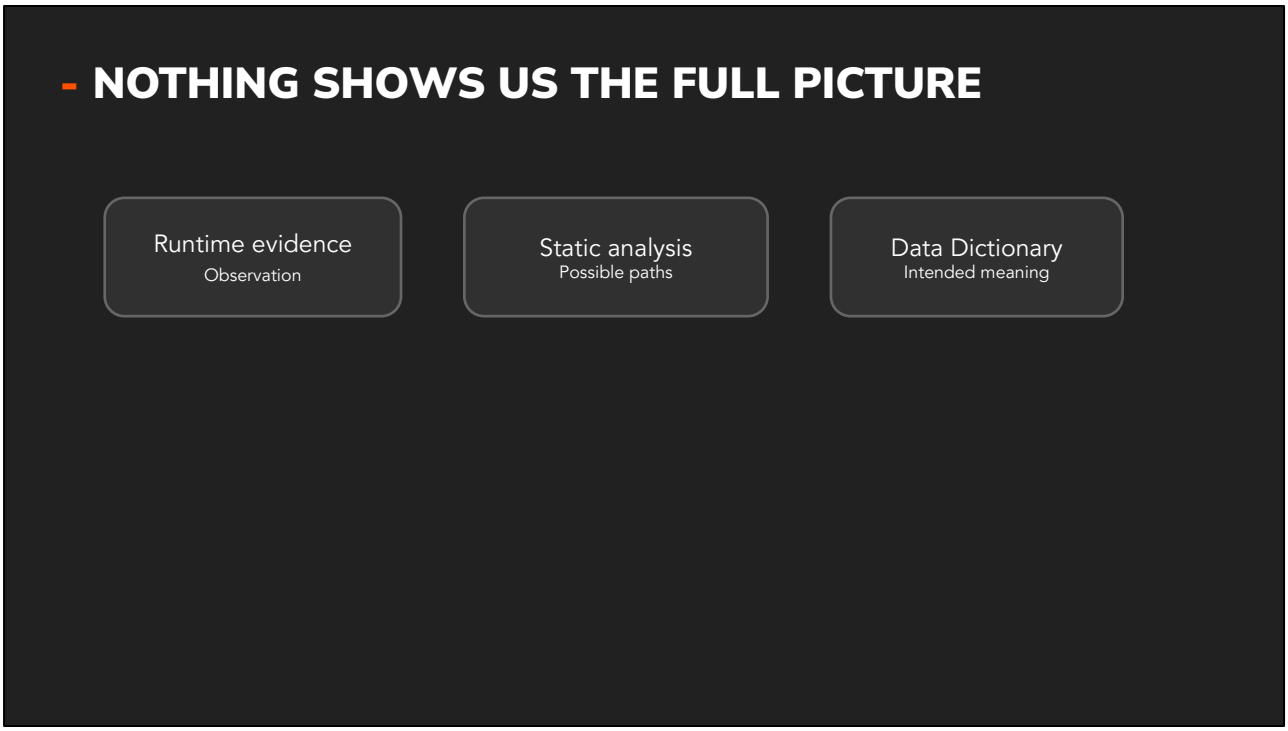
“Table X was queried”

to:

“This PHP entry point crossed this COBOL program and executed this query shape against Table X.”

But A high-coverage seam is also a high-volume seam.

To not kill our self we skipped per-row fetch events, suppressed known high-frequency fingerprints, sampled traces, and bounded attributes.



Now let’s have a look on the Code and runtime.

Runtime tells us what happened during the window we measured.

It cannot prove that an unobserved path is dead. A monthly job or recovery path may simply not have run that week.

Static analysis has the opposite limitation. It finds possible relationships, including paths that never execute.

We built language-specific extractors because one parser could not describe this system.

On the COBOL side, we resolve program sections, `CALL` statements, copybook usage, and embedded SQL.

On the PHP side, we resolve classes, includes, function calls, and ORM mappings where static resolution is possible.

Those extractors produce the same basic graph: repositories contain functions, functions call other functions, and functions access canonical database tables. Generated facades connect PHP callers to COBOL targets. Repository metadata

adds the responsible team.

Then there is the Data Dictionary.

In the COBOL world, this exists so programs can use shared definitions instead of every team independently deciding what a customer number, booking status, or date field should look like.

A simple example might be a short field such as `KDN`, where the dictionary defines its type, length, and business meaning once, and programs refer back to that shared definition.

Our analysis resolves 2,951 tables with Data Dictionary entries, more than 28,000 fields, and more than 30,000 field values.

But a value description still does not tell us who sets it, or which overnight process consumes it.

The runtime side starts with normalised SQL fingerprints. We associate them with canonical tables, call sites, and normalised stack-trace groups.

We currently have about 1,400 distinct call sites and almost 10,000 groups of runtime stack traces.

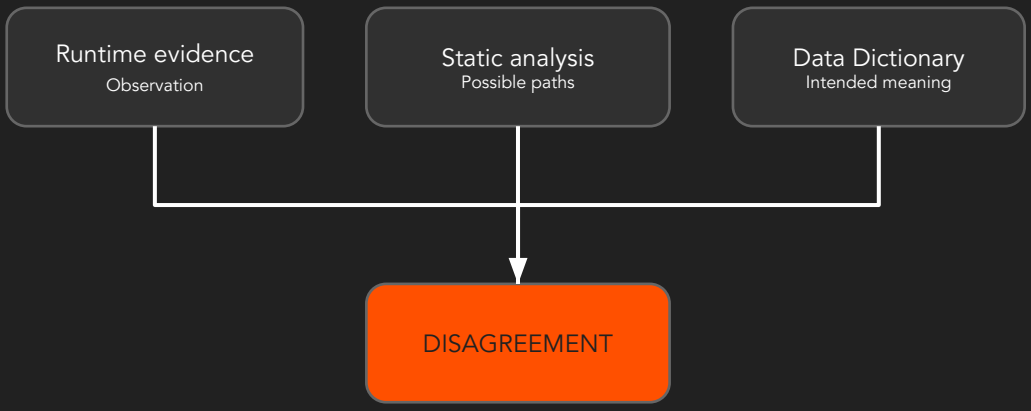
Think of it as mapping a city

We have identified roughly 1,400 intersections and nearly 10,000 routes through them.

And the interesting part is that these aren't routes someone drew on an architecture diagram.

These are routes we have actually observed in production.

- NOTHING SHOWS US THE FULL PICTURE



So we ended up with three witnesses.

Runtime gives paths exercised during a known window.

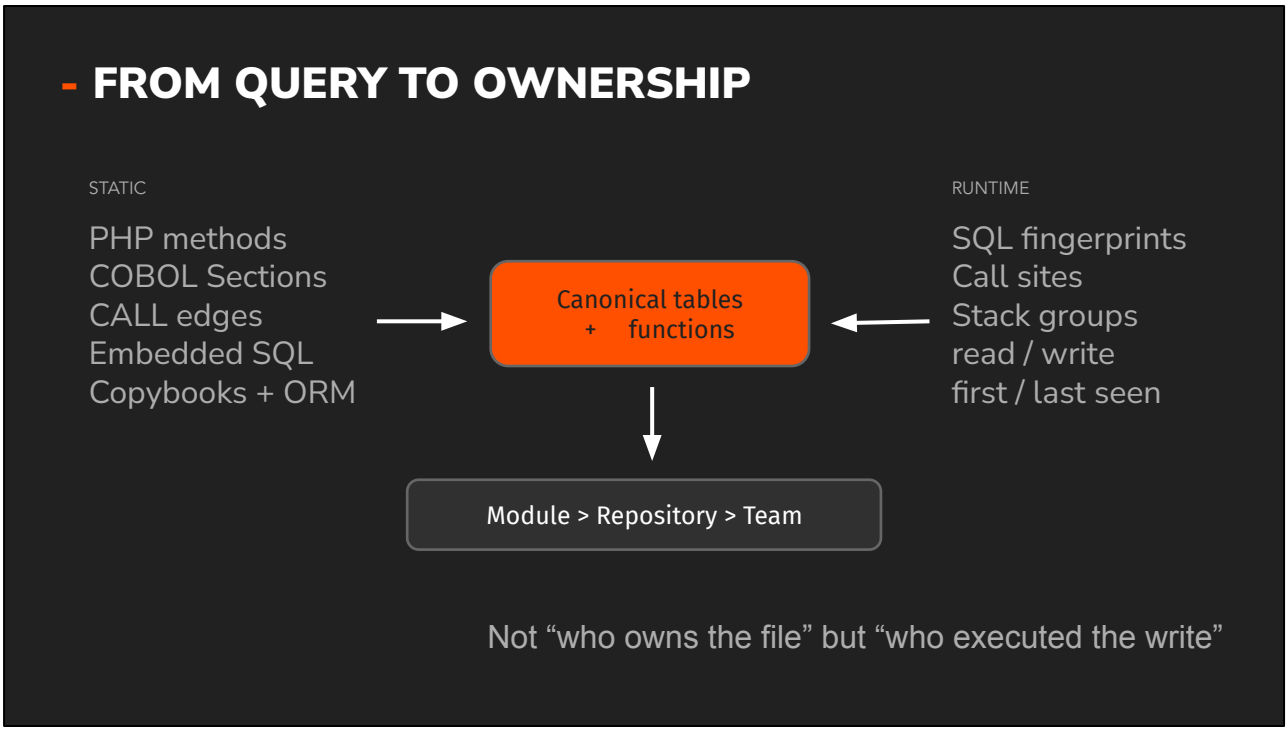
Static analysis gives possible paths.

The dictionary gives intended meaning.

The disagreement is often the useful part.

We currently have 2500 tables present in both static and runtime evidence, and 250 tables observed at runtime that are absent from the static table view.

Neither number proves that the remaining code is dead. It tells us where to investigate.



On the static side, language-specific parsers give us functions, call edges, cross-language facades, and function-to-table access.

On the runtime side, the SQL fingerprint identifies a stable query shape. Its call site and normalised stack lead us toward the executing module.

ORM mappings, aliases, and SQL parsing resolve different names back to one canonical table.

A repository-to-module mapping then connects that function to repository metadata and the responsible team.

The result is a chain we can inspect in either direction:

fingerprint to stack, stack to function, function to table, and repository to team.

The graph currently contains 525,000 functions, 710,000 call edges and 7,500 cross-language calls.

Those are static structure counts.

Runtime observations are stored separately and joined back to the graph.

A static edge means a call *could* happen.

A runtime observation means it *did* happen during the captured window.

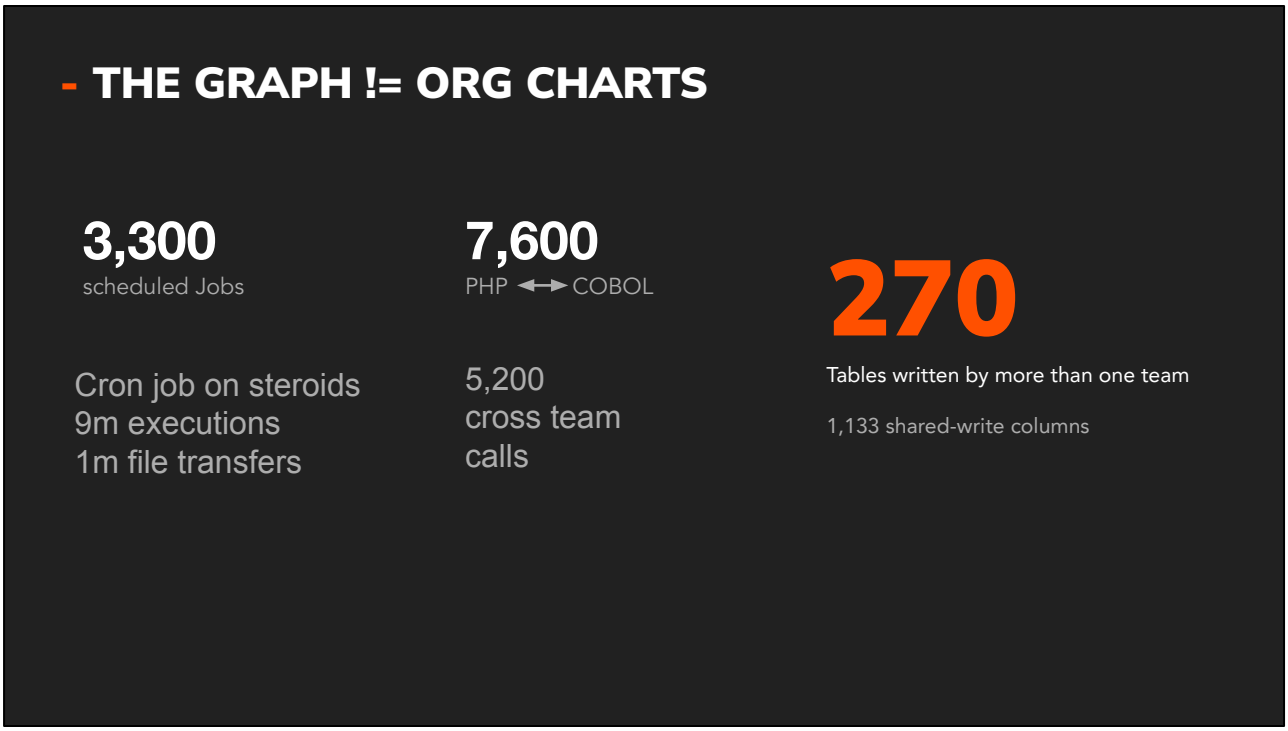
Neither one alone is an ownership decision.

BUT For the first time, we could answer questions down to the table column: who wrote this, through which path, and when did we observe it?

Research that previously could take weeks or month for one single Table could suddenly be answered in seconds.

But this was not an ownership database for its own sake. We need this Data so that we are sure which runtime paths covers which Business Usecase and had to survive .

The useful question became: who executed the write, through which path, during which observation window?



The resulting graph did not match the organisational picture.

Three findings changed how we talked about ownership.

First: the scheduler.

Our enterprise job scheduler defines 3,300 job steps. But just third quater of it where resolvable to PHP or COBOL calls, the left one quarter are P/SQL jobs.

Our scheduler triggers 9 Millions executions and triggers 1 Million file transfers per month, Its basically cronjobs on steroids.

Second: the language boundary.

Of the 7,600 PHP-to-COBOL and COBOL-to-PHP calls in the static graph, 5,776 cross team boundaries.

Third: shared writes.

We have write evidence for 2500 tables. In 270 of them, more than one team writes to the same table in some cases more than 10 teams are writing to the same column .

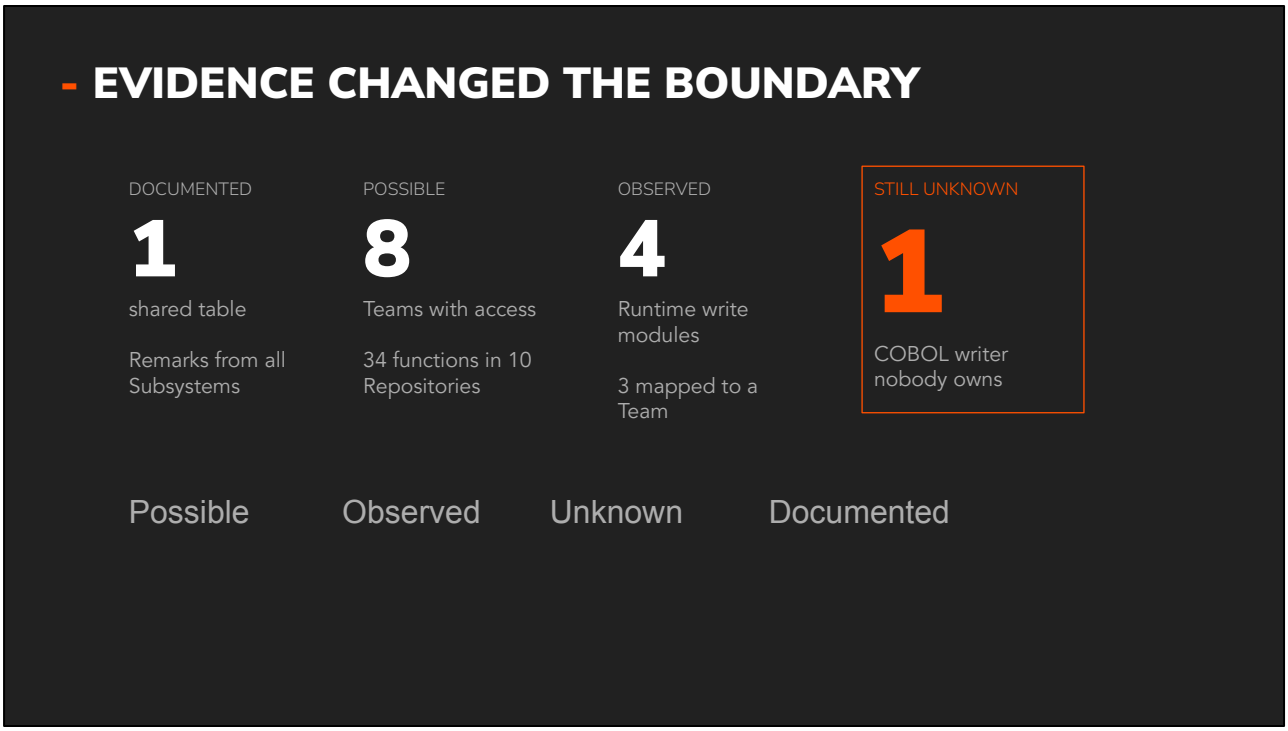
So Table Y “owned by team X” is not true in practice.

These numbers come from different sources.

the job steps come from scheduler configuration.

The write conflicts combine static ownership with runtime write evidence.

The bridge calls are static structure, not traffic.



Let me make that concrete with one real table, anonymised for this talk.

The Data Dictionary describes it as “remarks from all subsystems.”

One table. One sentence of documentation.

Static analysis finds 34 functions in 10 repositories that can touch it.

Eight teams have code with access: reads, inserts, updates, and deletes from both PHP and COBOL.

Runtime evidence narrows that down to what actually writes: four modules.

Three of those writers map to a team.

The fourth is a COBOL program that no team owns in our ownership data.

Nobody would have put it on the migration plan, because nobody knew it was a writer.

So this table cannot be migrated independently.

It was never an implementation detail of one team. It was an undocumented interface between several parts of the system, and one of its users had no owner.

And the data still does not decide ownership. It just tells us who has to be in the room.

That is the pattern for every boundary decision now:

documented, possible, observed, and still unknown.

The unknown part is usually where the risk is.

- WE COULD MAP THE SYSTEM

We had Photographs of our Systems but no Video

How hard could it be?

What is one Operation?

Who owns the context?

And what happens when telemetry fails?

Periodic SQL analysis helped us reconstruct the system.

But it could not give an operator a continuous trace of one action.

The analysis was a photograph. We needed a video.

We wanted to see one request cross from PHP into COBOL, and executing SQL, and run back into PHP and maybe execute another SQL.

And with, at that time just the logger in place which already sitting at the bridge and database-driver level, the obvious question came naturally:

“How hard could it be?”

There was no supported COBOL agent to install.

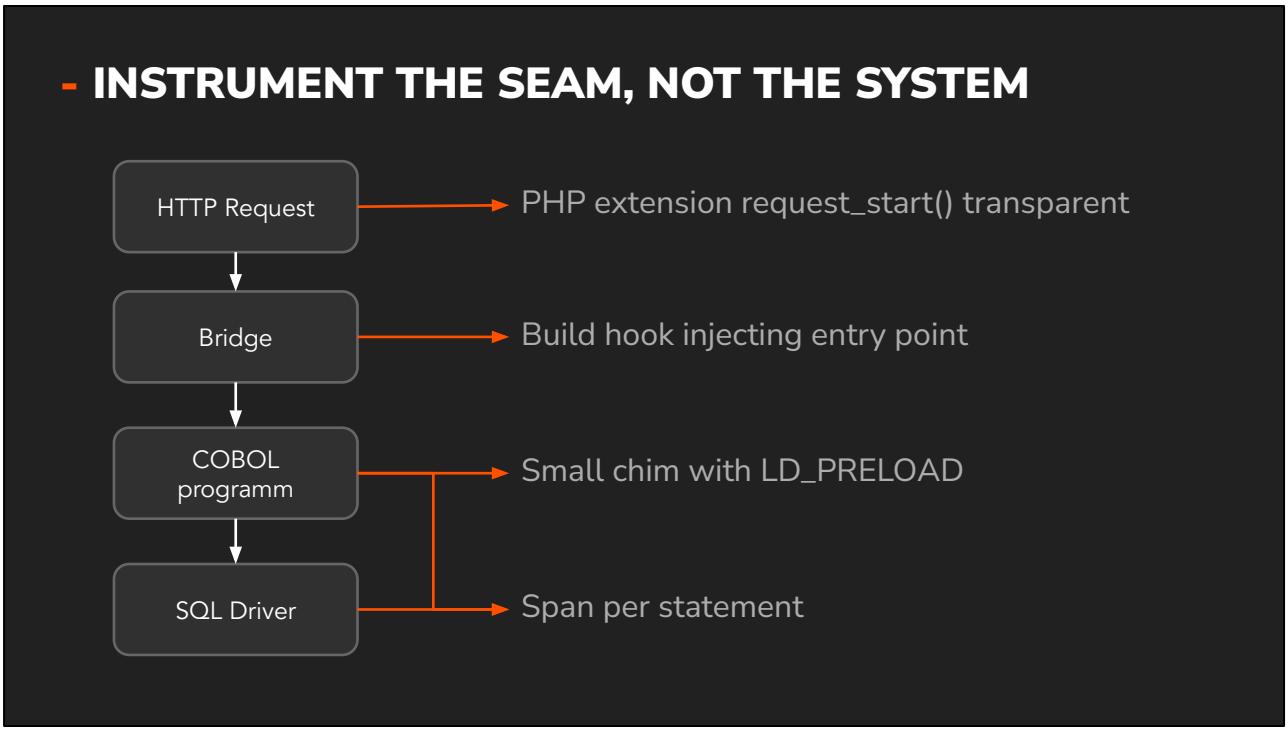
And once there was no SDK making the decisions for us, we discovered that an SDK normally hides three architectural questions:

What exactly is one operation?

Who owns its context?

And what happens when telemetry itself fails?

Those became the next three design problems.



Before we get to those three problems: where does our code actually run?

The unsupported runtime does not need a full modern SDK. It needs a small adapter to the modern world.

We instrumented where we had control.

There are four hook points, and each one needs a different trick because each one sits in a different kind of code.

Web requests start in our PHP extension. At request start, it creates the root span and picks up an incoming `traceparent` header if there is one.

The bridge is generated C code, powered by awk and some data dictionary entries.

So we did not touch it by hand so a build step injects a hook at every entry point of every converted COBOL program.

One build change, and every bridge call carries a span.

Batch has no request at all.

We load a small shim with `LD_PRELOAD` that intercepts the COBOL runtime's own module-enter and module-leave calls.

If the scheduler passes trace context in an environment variable, the batch root continues that trace.

And SQL goes through the driver, which is linked into web servers and standalone COBOL processes alike.

The driver looks up the OpenTelemetry functions at runtime. If the library is loaded, it emits a span per statement. If not, nothing happens.

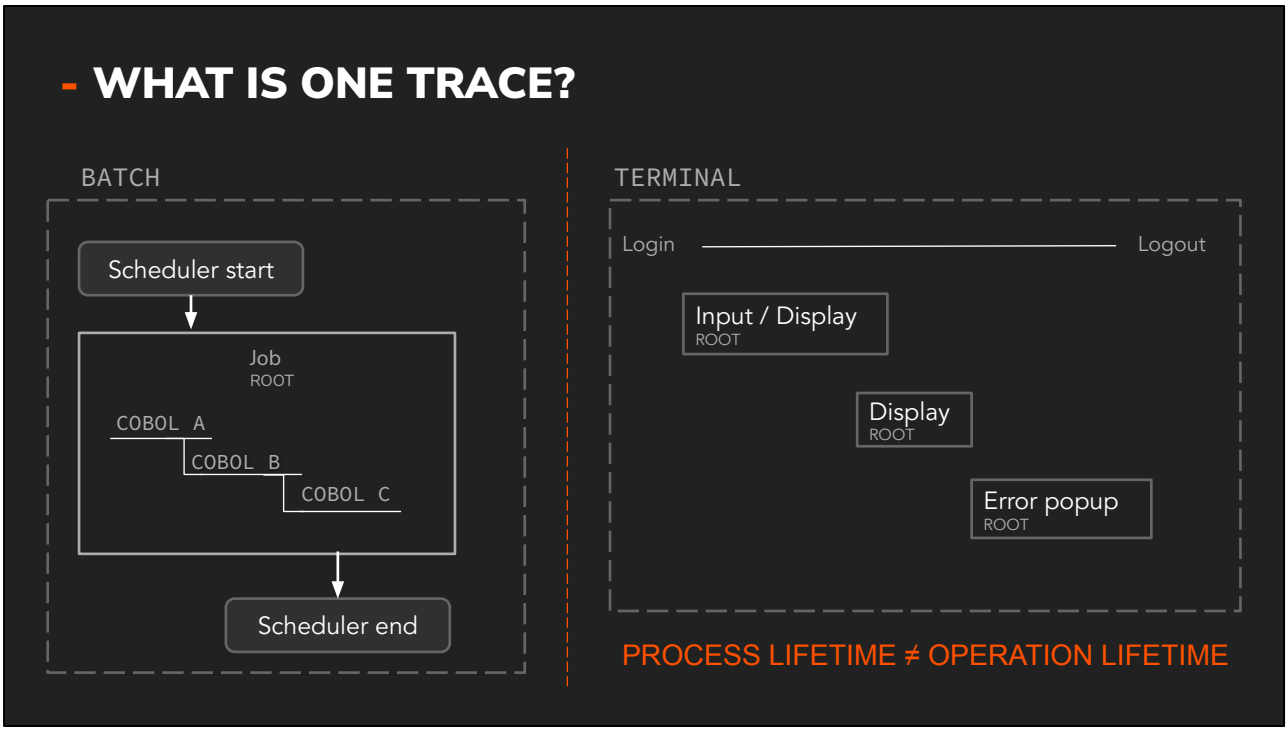
Same binary, no hard dependency.

Is that elegant? No.

But every hook sits on a boundary the work cannot avoid, which was the whole point of the seams.

Span creation and context propagation live in C.

That avoids editing hundreds of COBOL programs and gives us one implementation point for hundreds of bridge calls.



Now back to the first problem. It was deciding what one trace represents.

A batch process has a natural outer boundary:

the scheduler starts one execution, and the job ends.

That does not mean every record needs a span.

But one long bar for the whole job tells you nothing either.

So the job is the root span, and every COBOL program it calls becomes a child span.

That is the level where an operator can say, “This step got slow.”

A terminal process is different.

It may remain alive for an entire login session while the operator performs many unrelated actions.

Our instrumentation creates one root span for each meaningful interaction handled by `poprun`: input and display, display-only, or an error popup.

Input spans include the time spent waiting for a person, so we mark them as blocking rather than application latency.

Otherwise every coffee break would show up as a latency spike.

A useful boundary has an observable start, an observable end, an operator-recognisable name, and bounded work.

Process lifetime is not operation lifetime.



The second problem was context ownership.

We used thread-local C state with explicit lifetimes.

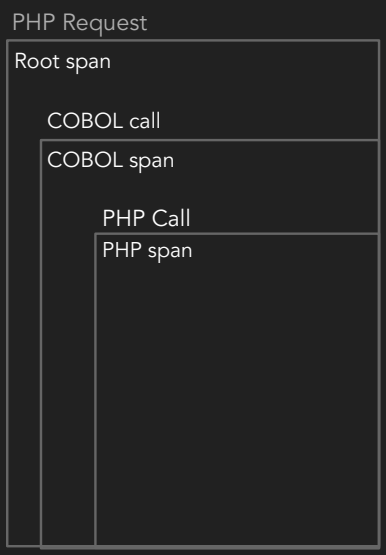
A PHP request owns its root span and a bounded COBOL-call stack.

At the bridge, a COBOL operation becomes a child of the active PHP span.

When COBOL calls PHP, that operation becomes a child of the active COBOL span.

Shutdown clears the state.

- WHO OWNS THE CONTEXT?



THREAD-LOCAL C STATE

PHP request
↓ owns
[root]

Bridge
↓ propagates
[COBOL]

Callback
↓ propagates
[PHP]

ABORT

STOP RUN / exit() / die()
↓
×
no return

C destructor
↓
close remaining spans mark error

A WRONG PARENT IS WORSE THAN AN HONEST GAP

Shutdown sounds simple until you meet COBOL.

A batch program can end with `STOP RUN`, which calls `exit()` directly.

So no return path and no cleanup in our code.

PHP code called from COBOL can do the same with `exit()` or `die()`.

Without help, every open span would simply vanish.

So a C destructor runs at process exit and flushes whatever is still open, marked as an error.

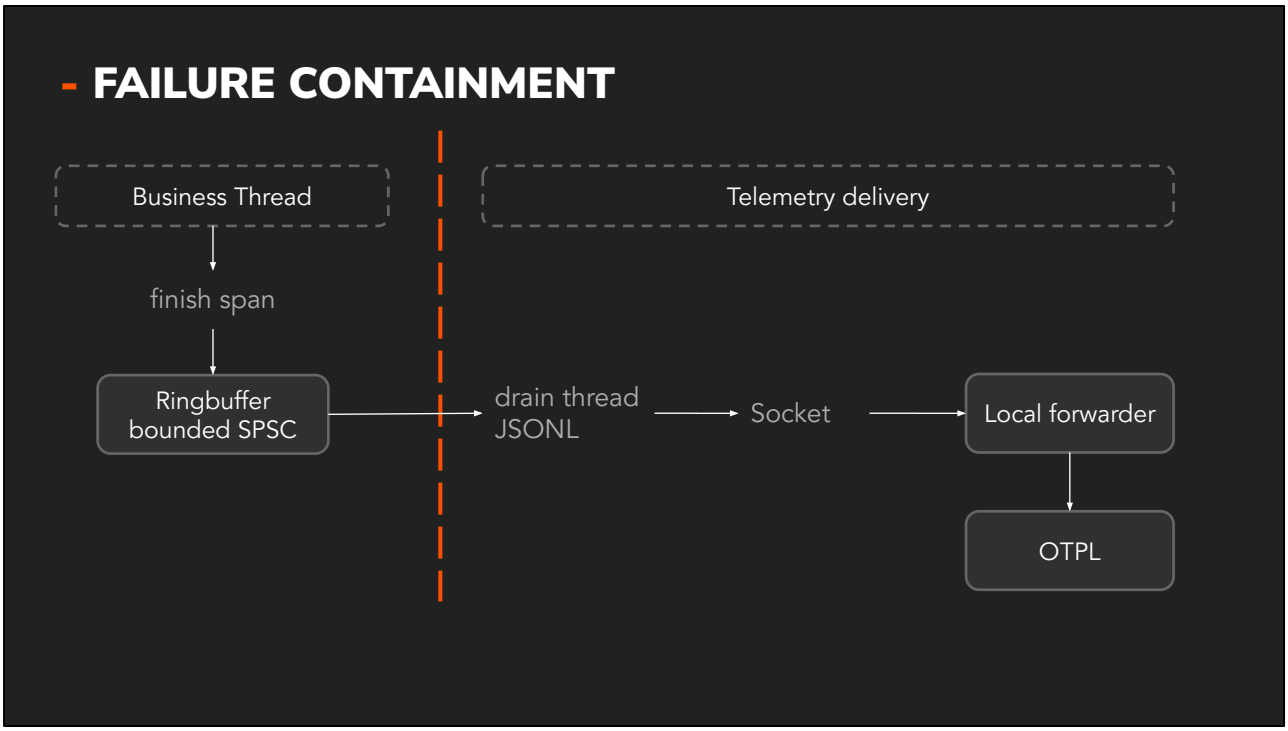
The mechanism matters less than the ownership model.

Who sets the context?

Who copies it across a boundary?

Who clears it?

A connected trace with the wrong parent is worse than an honest gap.



The third problem was failure containment.

Our C implementation finishes spans into a bounded single-producer, single-consumer ring buffer.

A drain thread serialises them as JSONL and sends them over a local Unix socket to a forwarder.

Why a local socket and not OTLP straight out of the process?

Because then every COBOL and PHP worker would need a network client, with retries, timeouts, and TLS.

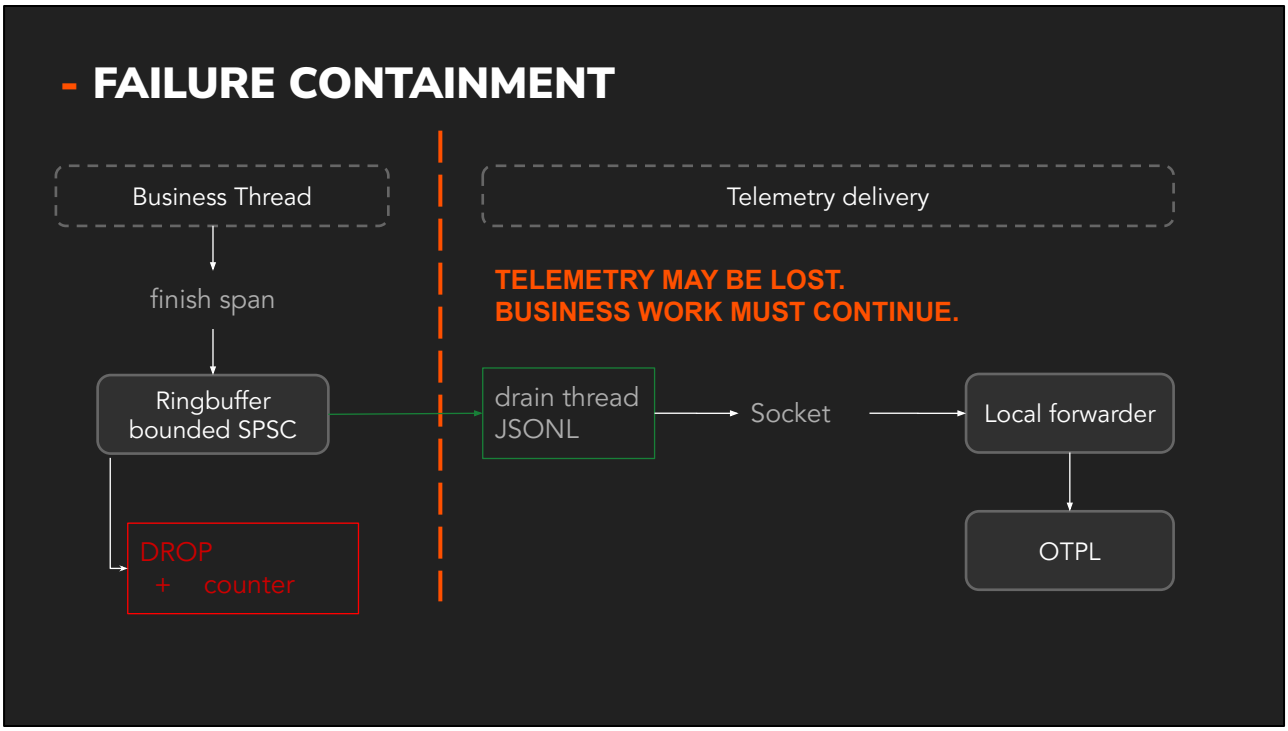
That is a lot of new ways to fail inside a process whose actual job is to book a rental.

So the business process only ever writes to a local socket.

The forwarder owns the network.

The buffer is lock-free.

The business thread writes, the drain thread reads, and they never wait for each other.



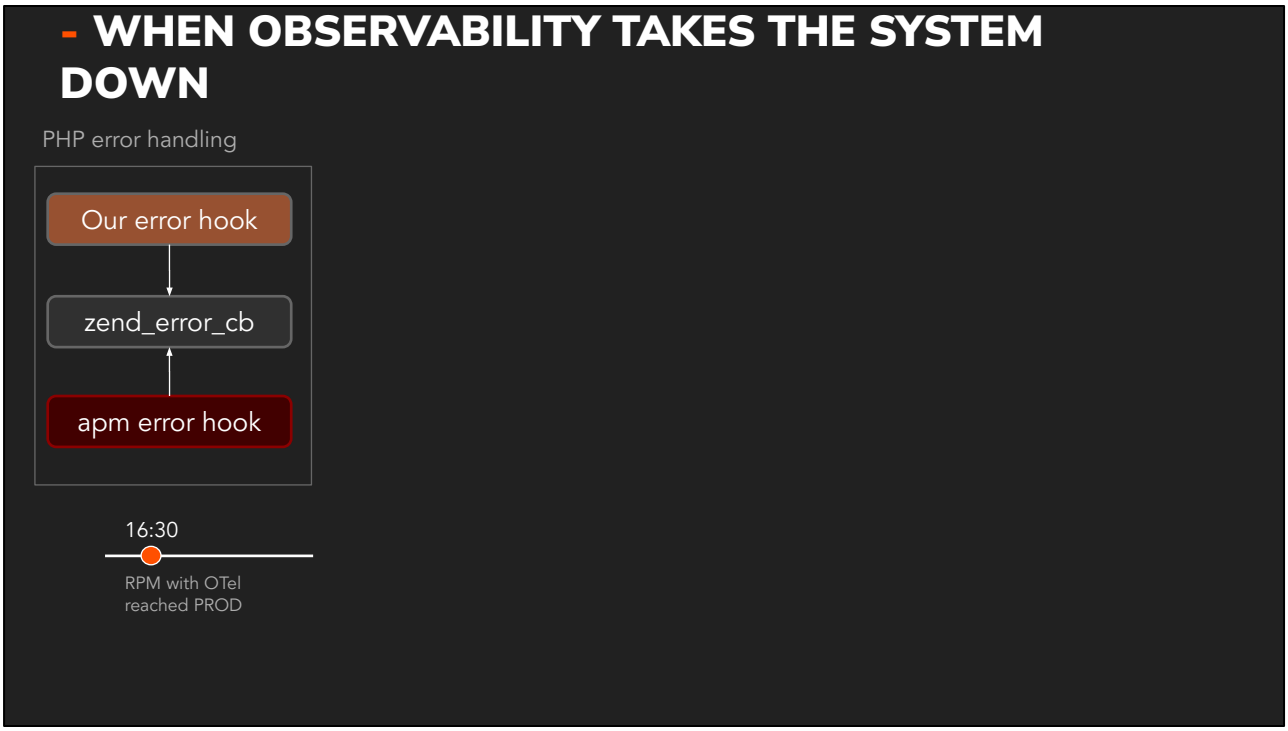
When the buffer is full, the span is dropped on the spot, and the drop is counted.

Losing telemetry is allowed but Losing it silently is not.

If the buffer is full, the forwarder is absent, or a socket write fails, telemetry is dropped rather than blocking business work.

That is the principle:

Telemetry may be lost.
Business work must continue.



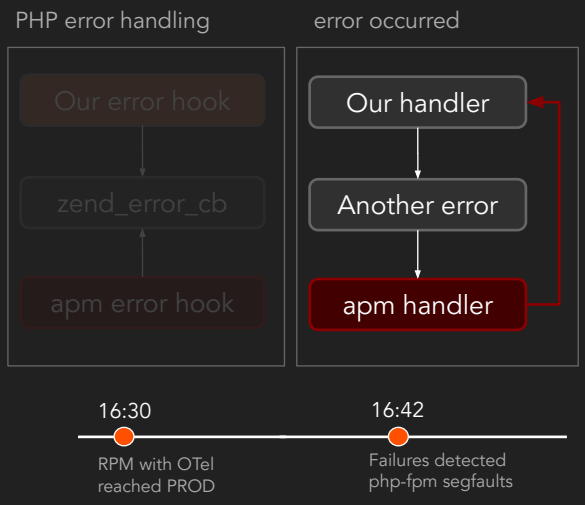
And then we learned what happens when that principle is not applied everywhere.

At about 16:30, a production RPM containing our OpenTelemetry work reached the fleet, because our build process is one of a kind.

That alone would have been embarrassing but survivable.

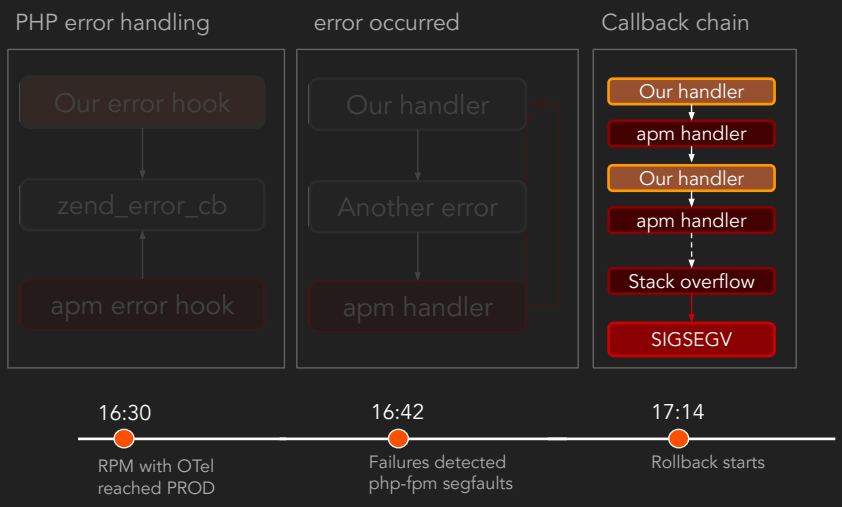
But our PHP error hook and Instana's error hook both participated in PHP's single mutable `zend_error_cb` chain.

- WHEN OBSERVABILITY TAKES THE SYSTEM DOWN

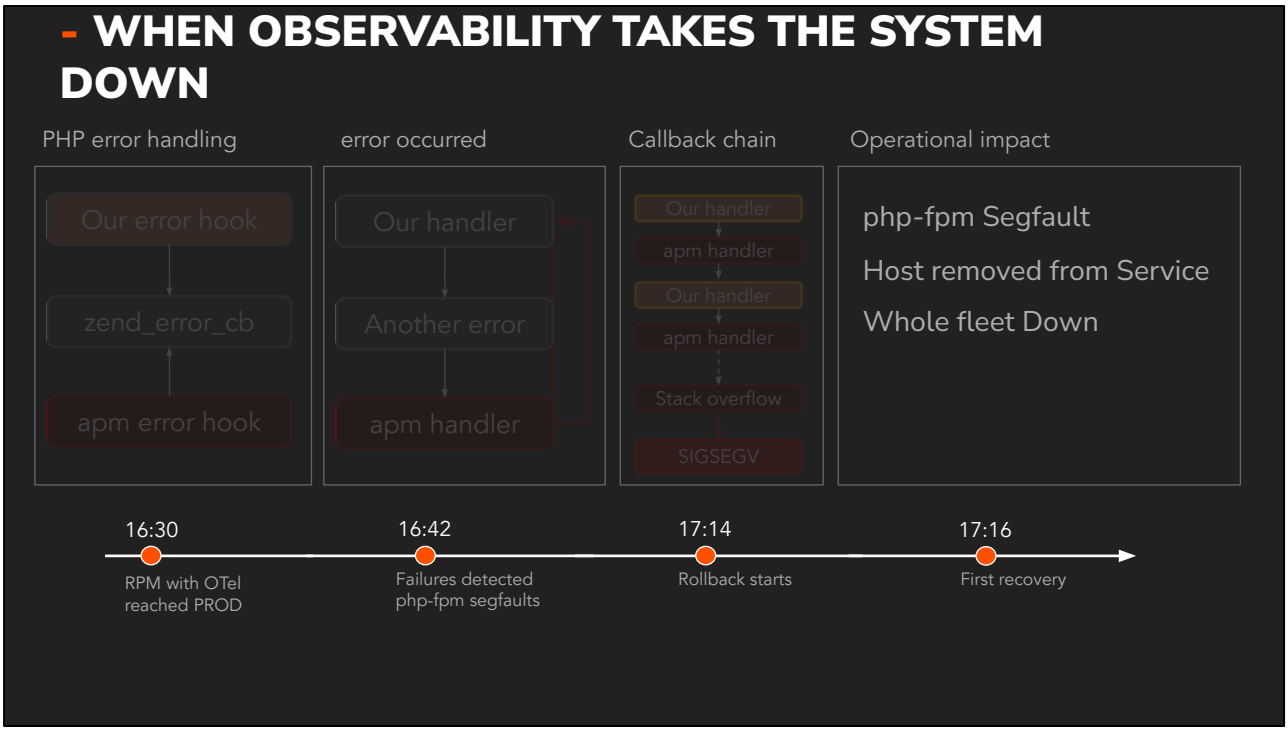


Under a production error pattern, handling an error triggered another error and re-entered the chain.

- WHEN OBSERVABILITY TAKES THE SYSTEM DOWN



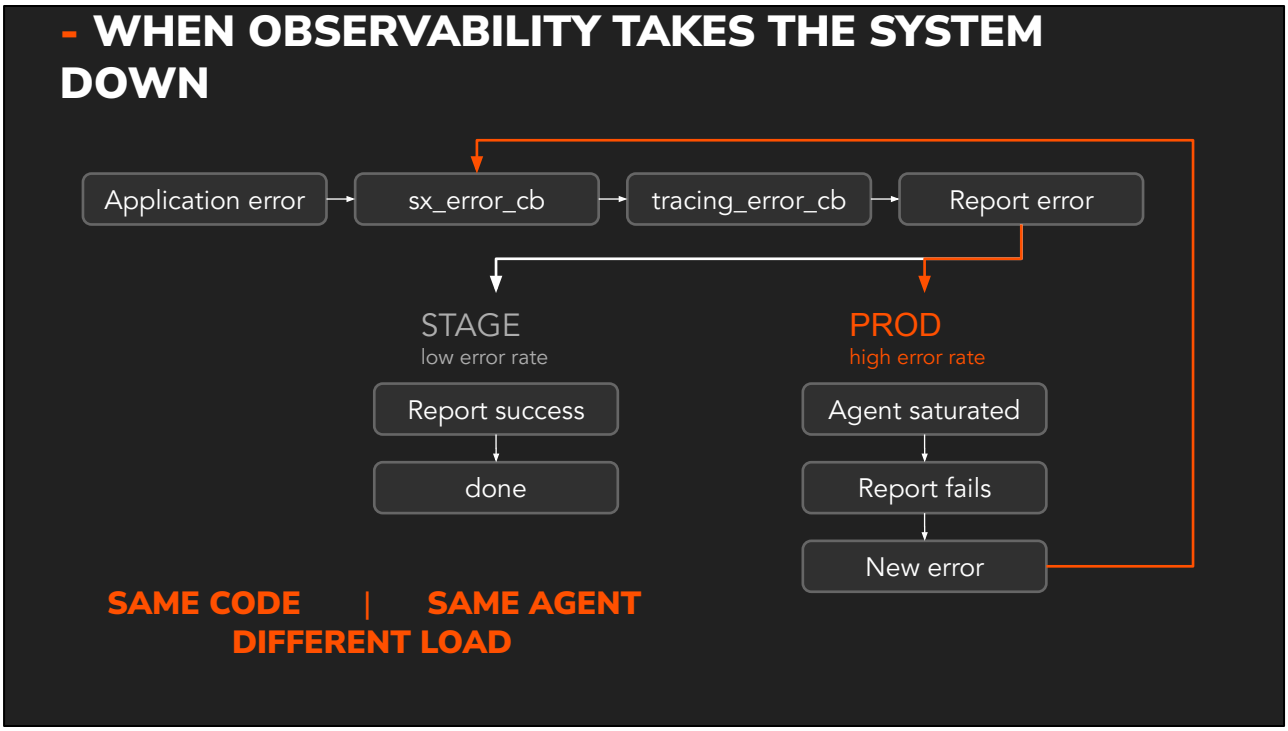
The stack grew through repeated callbacks until PHP-FPM workers segfaulted.



One host recorded 200 SIGSEGVs in 39 minutes, and failed health checks took service out and that happened in the whole fleet.

Great… reeeaally great.

We rolled the RPM back. And two minutes later first services are recovering.



Here is the chain in detail.

An application error goes to our handler, `sx_error_cb`.

Our handler passes it on to the comercial handler, `tracing_error_cb`.

Both sit in the same global hook, and PHP only has one of those
.

On a normal day, that is fine. Each error is handled long before the next one arrives.

Under production load, the error rate crossed a threshold.

The agent saturated and could no longer send.

Failing to report an error is itself an error, so the handler raised a new one.

That new error went straight back into the same chain.

No single call was wrong.

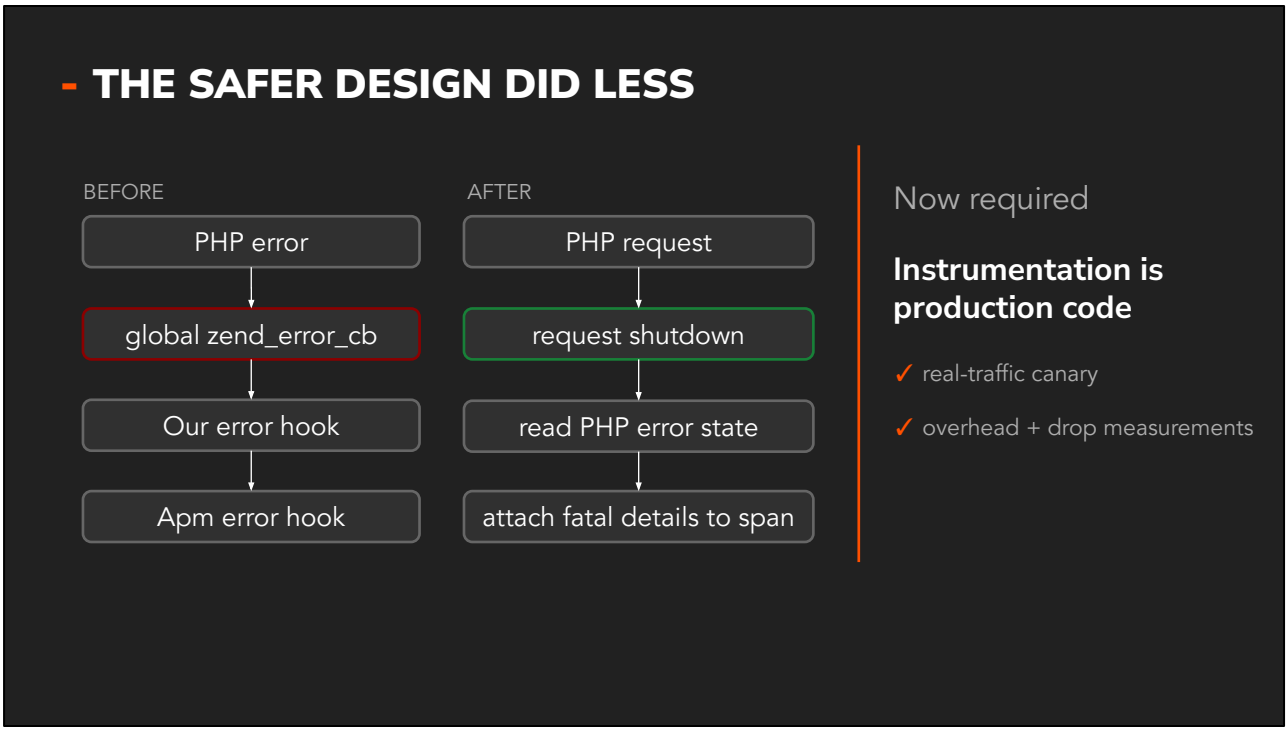
The problem was a cycle with no exit, running inside a hook that every request

shares.

So what looked from the outside like “PHP-FPM is randomly crashing” was actually two observability implementations enthusiastically observing each other to death.

But here is the point - Staging ran exactly the same setup for months, and it never showed this behaviour.

It was in the combination under production load.



We stopped participating in the global callback chain.

At request shutdown, we read the error state already maintained by PHP and attach fatal-error details to the request root span.

We accepted less precise attribution to avoid competing for a global hook.

The safer design did less And that was exactly the point.

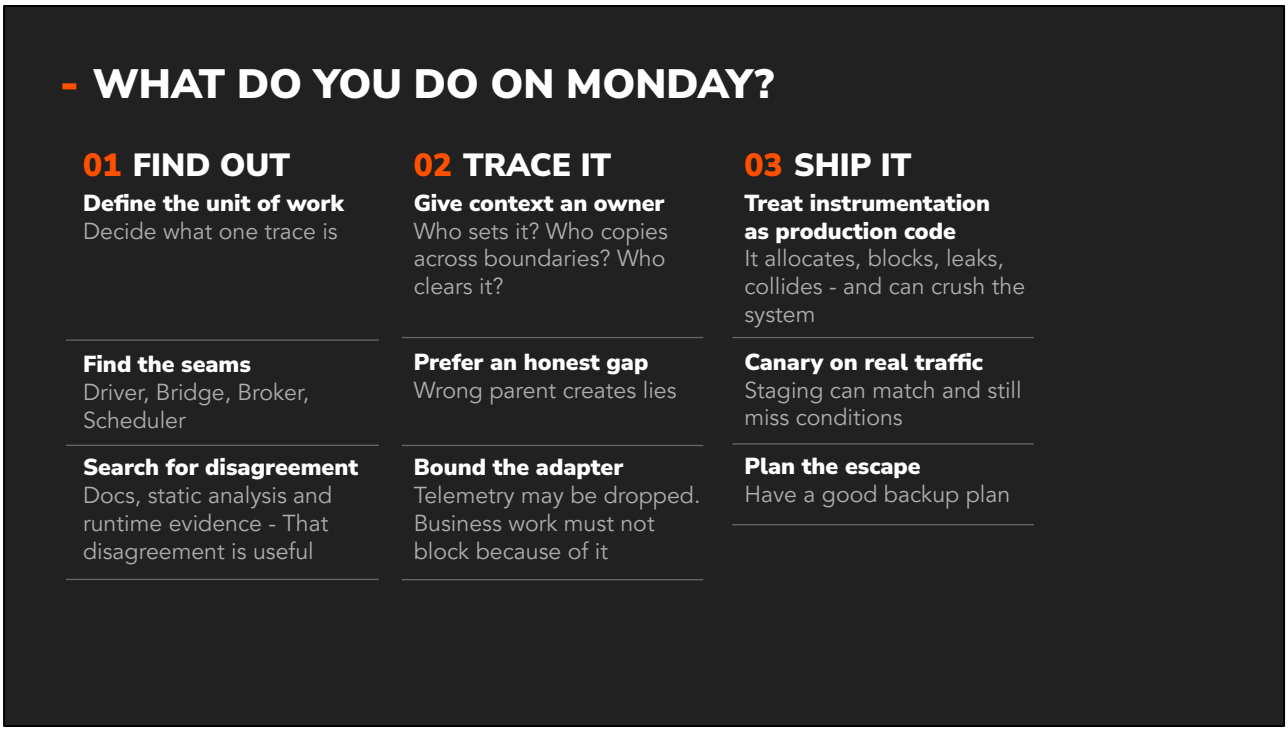
Instrumentation is production code. Duhh

It allocates memory, can block, leak context, collide with another agent, or crash the process.

Our rollout requirements now include a canary on real production traffic, application-health checks, overhead and drop measurements, a kill switch, and a tested rollback.

Staging with the same agents was not enough, because staging does not produce production error rates.

And I am now unhealthily paranoid about our release and branch handling.



So what do you do on Monday if your own unobservable system is a Java 1.4 batch worker, a proprietary runtime, or a framework whose vendor stopped answering emails?

First: do not start by writing an agent.

Start by finding things out.

Define the unit of work before you create a single span.

Our terminal sessions ran for hours. One session is not one trace. One action is.

Then find the seam the work cannot avoid.

For us, it was the database driver and the language bridge.

For you, it might be a JDBC driver, a message broker, or a scheduler wrapper.

Every real action has to cross something.

Observe that.

And let the witnesses disagree.

Documentation, static analysis, and runtime will tell you three different stories.

The disagreement is the finding.

Remember the table with four writers, one of which nobody owned.

Second: trace it.

Give context an owner and a lifetime.

Who sets it?

Who copies it across the boundary?

Who clears it?

A trace with the wrong parent is worse than no trace.

And bound the adapter.

Telemetry is allowed to get lost.

Business work is not.

Third - and this is the one I learned the hard way - ship it like production code.
Because it is production code.

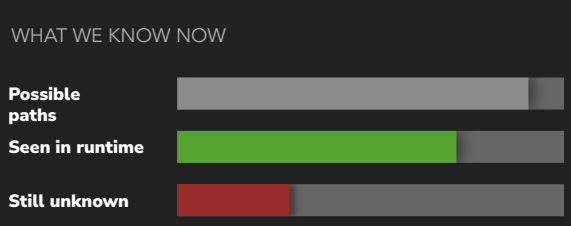
Staging ran the same agents as production and saw nothing.

Canary on real traffic, watch health signals,

keep a kill switch, and test your rollback before you need it.

- WE DECIDE WITH EVIDENCE NOW

We started because we wanted to know what we could switch off



"We patched the db driver" is rarely comforting

But it was the seam every real action had to cross

QUESTIONS?

We started this because we wanted to know what we could switch off.

But We are not done, we now know which paths are possible, which we have actually seen, and which are still unknown.

We make decisions with that evidence now, instead of with hope.

And, "we patched the database driver" is not a sentence that makes an SRE or SDE sleep better.

But for us, it was the one place every real action had to pass through, which made it the best observation point we had.

That's it from me folk.

I'm happy to take questions now, I'll be around after the talk and at the AMA as well.