
Core C++ 2026
20-21Oct. 2026 :: Tel-Aviv
From Unit to Production:
3 Layers of Performance Measurement
Alex Cohn



Priority

Alex Cohn

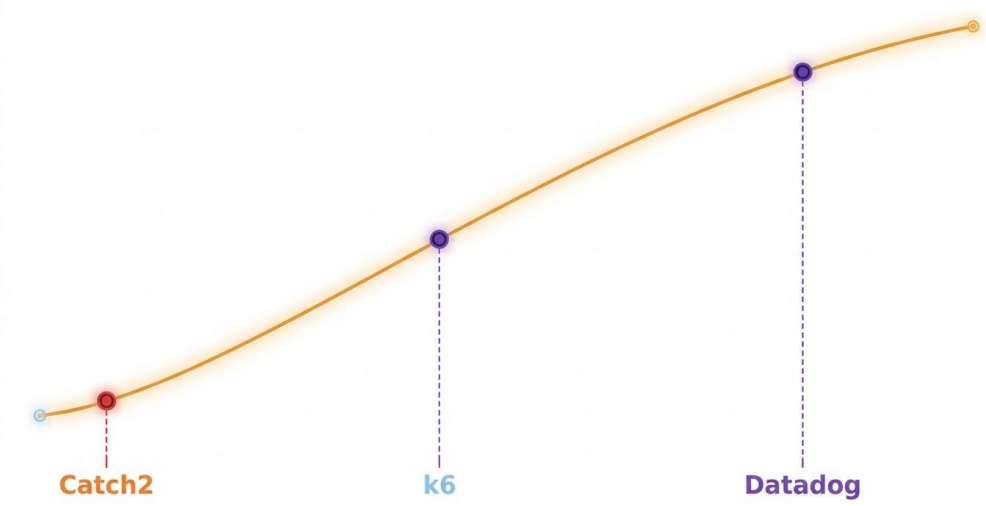
@sasha.cohn

Improving performance of AWS C++ services at ***Priority Software***

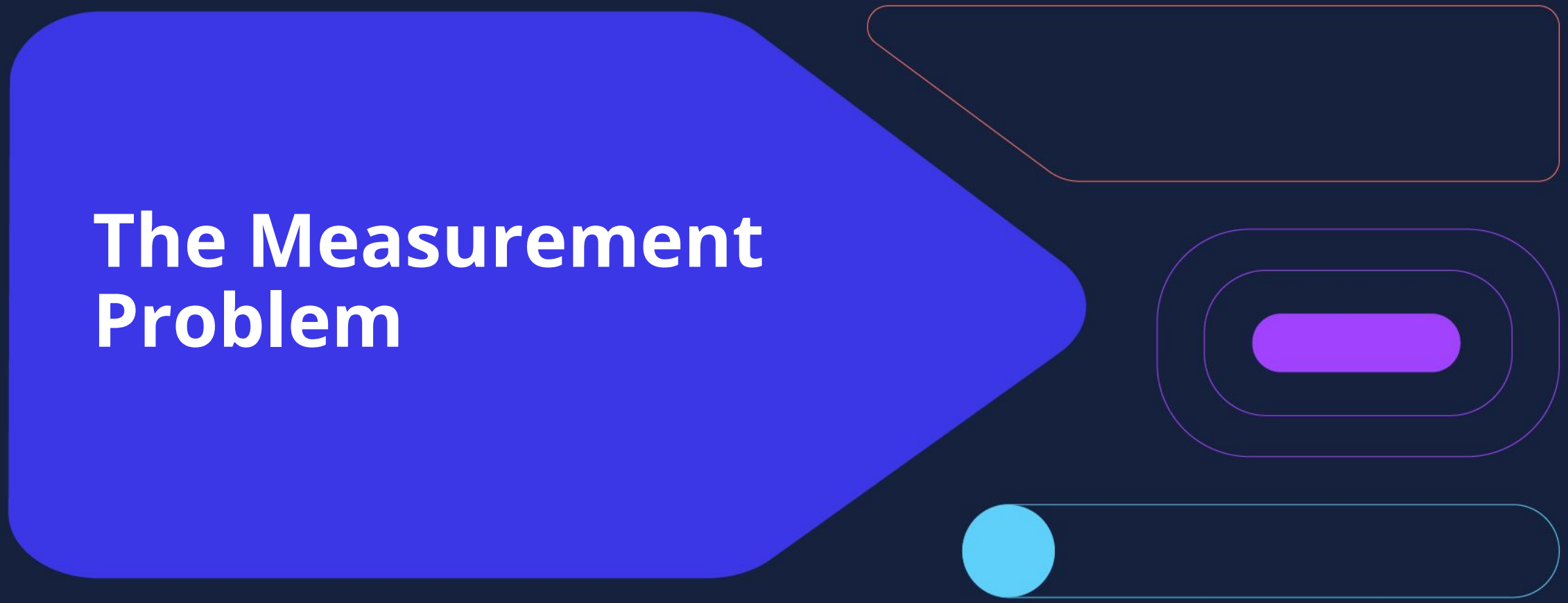
I take it personally if it's slow

A Three-Layer Approach

1. The Measurement Problem
2. Layer 1: The Code — Catch2
3. Layer 2: The System — k6
4. Layer 3: Production — Datadog
5. Putting It All Together



The Measurement Problem



"It's slow."

- Not a bug report
- Not a ticket
- Not something you can fix
- By the time the customer complaint reaches you, all evidence is gone

Why Cloud Performance Is Different

- Shared physical resources — your neighbors matter
- Elastic scaling — the system you test is not the system you run
- Distributed by default — latency hides in unexpected places
- Pay-per-use — inefficiency costs money, not just time
- Cold starts, retries, throttling — invisible until they hurt

- You cannot measure what you cannot see
- You cannot fix what you cannot measure

A Three-Layer Measurement Strategy



Layer 1: Code

Catch2 BENCHMARK

Micro-level precision

Function / unit scope

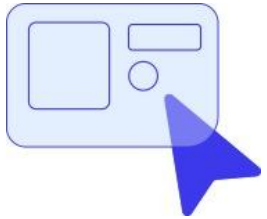


Layer 2: System

k6 Load Testing

End-to-end flows

Realistic concurrency



Layer 3: Production

Datadog APM

Real workloads

Distributed traces

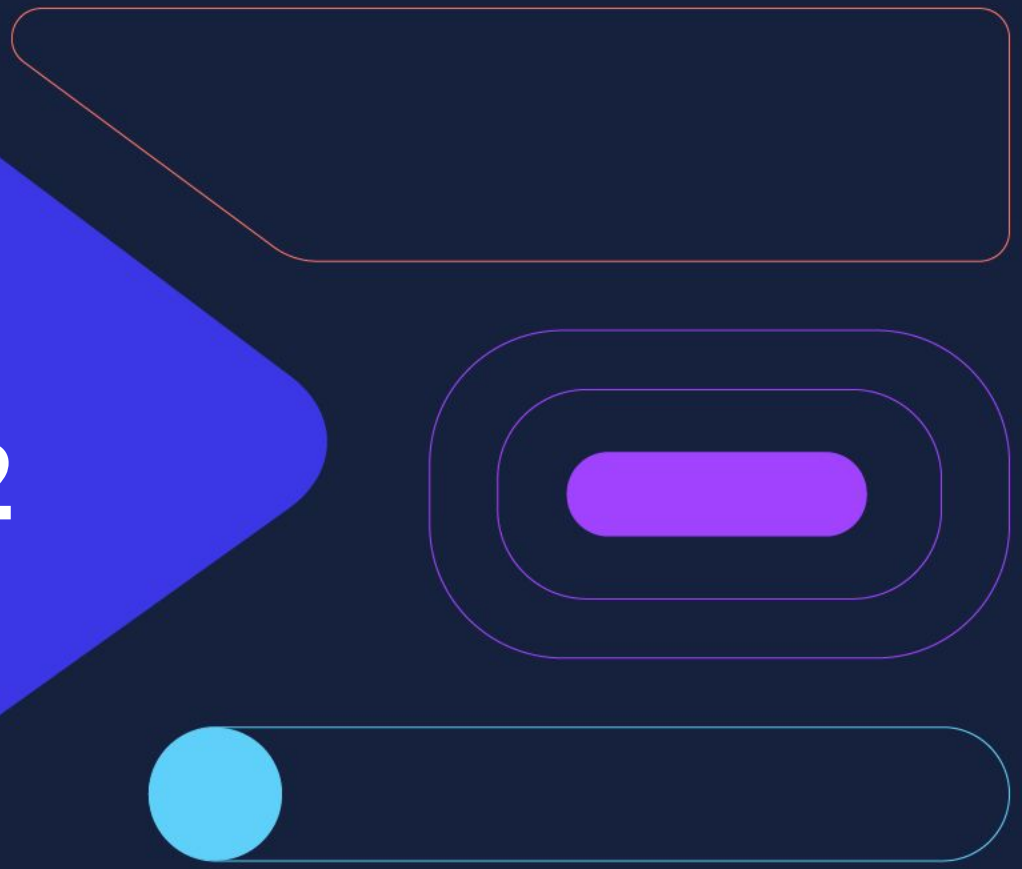


The Bridge

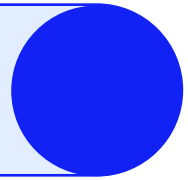
Layers agree: confidence

Layers disagree: insight

Layer 1:
The Code — Catch2



Catch2: Beyond Unit Testing



Catch2's hidden gem: BENCHMARK

Catch2 BENCHMARK lets you:

- Measure execution time of C++ functions
- Run multiple iterations automatically
- Report mean, std dev, confidence interval

No external harness. No extra tools. Just C++.

```
#include <catch2/benchmark/catch_benchmark.hpp>
```

BENCHMARK: The Basics

```
#include <catch2/benchmark/catch_benchmark.hpp>
TEST_CASE("create XML") {
    auto payload = buildTestPayload();
    BENCHMARK("fixed string") {
        return create_xml(payload); // avoiding pitfall
    };
    BENCHMARK("random data") {
        return create_xml(buildTestPayload());
    };
}
```

Microbenchmark Pitfalls

1. **The compiler is too smart:**
   - Dead code elimination — you benchmark a no-op
   - Constant folding — result computed at compile time
2. **Wrong level of isolation:**
   - Too-small function → many iterations → measuring overhead, not your code
   - The function may be inlined in your production binary
3. **Cache effects distort everything:**
   - First call: cache miss. Second call: cache hit

ADVICE: Use the same compiler options as in production, link the actual production .DLL (or .SO).

BENCHMARK Output: Reading the Results

```
test_xml_sources.exe is a Catch2 v3.12.0 host application.
Run with -? for options

-------------------------------------------------------------------------------
create XML
-------------------------------------------------------------------------------
C:\sources\alex.cohn\devel\util\tests\xml\test_catch_xmlwriter.cpp(175)
...............................................................................

benchmark name                       samples       iterations    est run time
                                     mean          low mean      high mean
                                     std dev       low std dev   high std dev
-------------------------------------------------------------------------------
fixed string                               100             2     4.1174 ms
                                    12.988 us     12.789 us     13.591 us
                                   1.60497 us    651.018 ns    3.49747 us

random data                                100             1     9.2281 ms
                                    96.463 us     93.538 us     100.679 us
                                   17.6958 us    13.2342 us     27.2771 us

random data advanced                       100             1     2.9983 ms
                                    59.142 us     57.104 us     61.432 us
                                   11.0145 us    9.36582 us     13.699 us

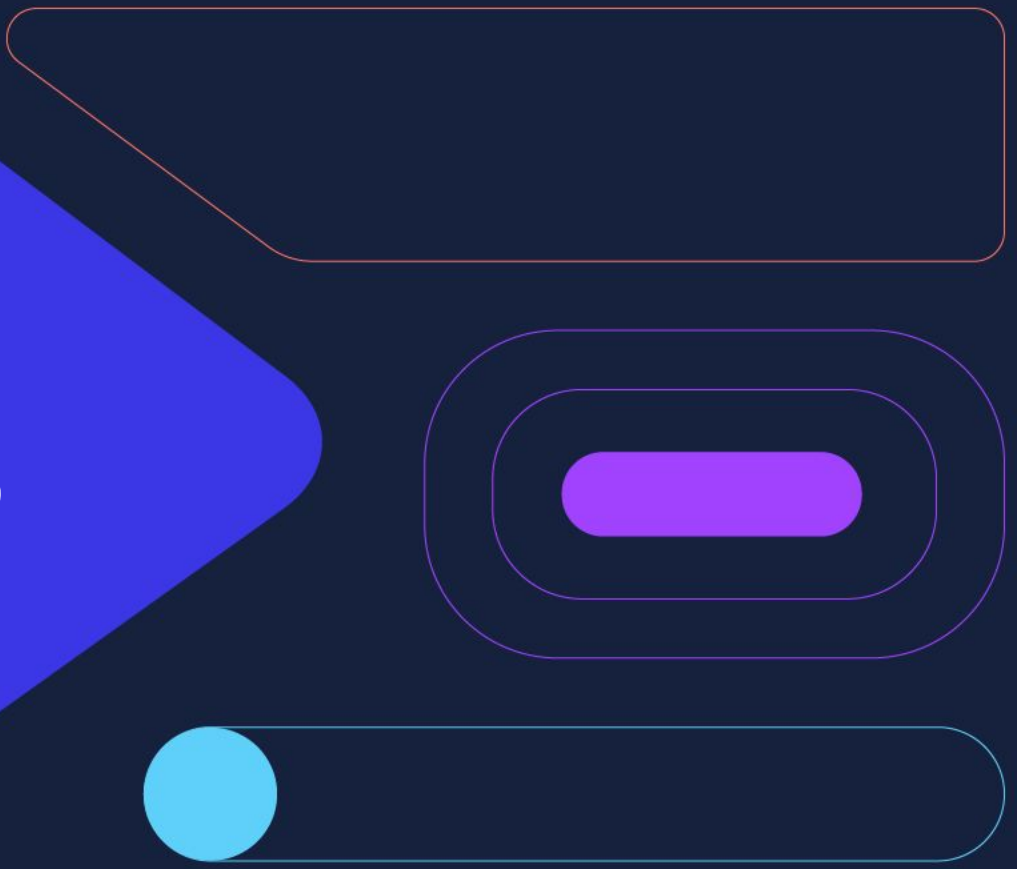

===============================================================================
All tests passed (20891 assertions in 1 test case)
```

BENCHMARK_ADVANCED: more control

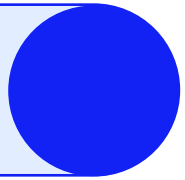
```
BENCHMARK_ADVANCED("random data advanced")
        (Catch::Benchmark::Chronometer meter)
{
    auto payload = buildTestPayload(); // setup — not timed
    meter.measure([&] {
        return create_xml(payload);   // only this is timed
    });
}
```

⚠️ If you can see your setup in a profiler, it belongs outside **meter.measure()**.

Layer 2:
The System — k6



k6: Load Testing by Grafana



- **Tool for load testing of Web services performance**
  - Best used in a “staging” environment
  - Scripts are all real JavaScript code: *version controlled, and reviewed*
  - Runs locally, in CI/CD pipelines, or in Grafana Cloud.

- **What k6 gives you:**
  - Simulate hundreds of concurrent virtual users
  - Ramp up / steady state / ramp down scenarios
  - Percentile distributions: p50, p95, p99: high variance expected, tails matter
  - CI integration: fail the build if goal not achieved (e.g. p99 > 500ms)

Scripting a Realistic AWS Scenario

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  stages: [
    { duration: '1m', target: 50  },  // ramp up
    { duration: '5m', target: 50  },  // steady state
    { duration: '1m', target: 0   },  // ramp down
  ],
  thresholds: {
    'http_req_duration': ['p(95)<500', 'p(99)<1000'], // ← this fails the CI build if breached
  },
};

export default function () {
  const res = http.post(WORKFLOW_URL,  JSON.stringify(payload), HEADERS); // ← this is what is measured
  check(res, { 'status 200': r => r.status === 200 }); // ← records a metric but doesn't fail the test
  sleep(0.1); // ← 100 ms think time, not part of http_req_duration, avoid back-to-back
}
```

The “mean” Is a Liar

- p50: 120ms — feels fast
- p95: 890ms — 1 in 20 users waits nearly a second
- p99: 4200ms — 1 in 100 users ready to complain publicly
- mean: 145ms — looks fine in your dashboard

⚠️ **The mean hides everything that matters!**

Rules:

- Set SLOs on p99, not average.
- Use k6 thresholds to enforce them automatically.
- If p99 spikes and mean stays flat: you have outliers.
- *Outliers are user complaints waiting to happen.*

🤦 The Tails That Matter

Catch2 benchmark is good (146 ms with high confidence)

Wrapped as a Web API, tested with k6 with 50 VUs:

The k6 median time was very good, 238 ms

Red flag 🚩 *min* was … 50 ms

The caveat: k6 will happily count the cases when the API failed

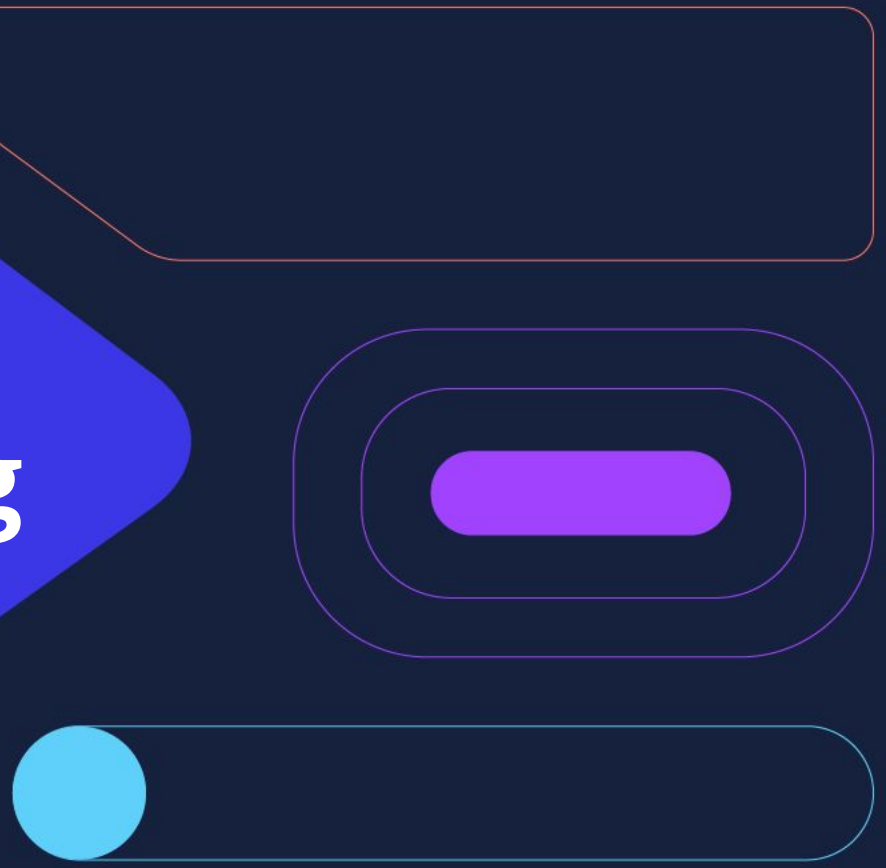
Another Web API …

Median 75ms, 🚩 *p95* 3 sec.

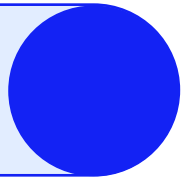
What could go wrong?

Once in a while, the common cache file got corrupted.

Layer 3:
Production — Datadog



Datadog: Full-Stack Observability



Three pillars — linked by a single trace ID:

- Metrics — custom counters, histograms, gauges
- APM — distributed tracing: follow one request across all services
- Logs — generated during that exact request

What Datadog gives you that k6 cannot:

- Real user traffic — not synthetic load
- End-to-end trace across system and custom services
- Correlation: slow request → which service → which function

For C++: no auto-instrumentation. You write it. It’s not hard.

APM: Traces and Spans

Flame graph of one Web request: Six services involved. One **trace_id**.



Datadog Tracer in C++

```
// git clone https://github.com/DataDog/dd-trace-cpp
void processRequest(const Request& req) {
    auto span = tracer.create_span("http.request");   // ← root span
    span.set_tag(“user.id", req.user); // ← cardinality not a problem
    {
        auto child = span.create_child("RunQuery");   // ← child span
        child.set_service_type("sql"); // ← will show in yellow
        runQuery(req.query);
    }  // ← child ends here (RAII)
}  // ← root ends here
```

🤦 The Tale of a Missing Span

Datadog trace showed a 160 ms gap after the Web service received the request and before the C++ service began to handle it.

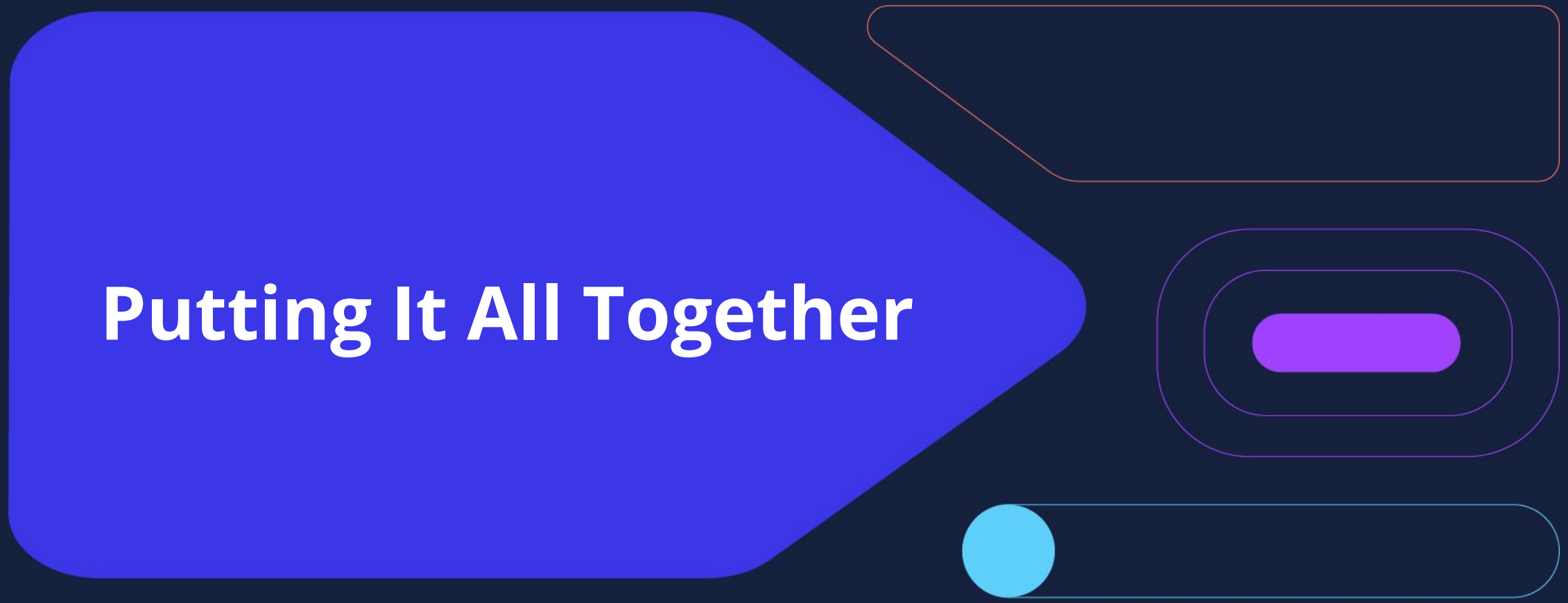
We suspected that it took the Web service too long to start gRPC connection.

No, the C++ service was responsible. To start processing the gRPC request, it had to verify that the request was valid. This verification launched an SQL query: 🪳, *please don't do this!!*

It was hard to detect because *Datadog* **trace_id** *was part of the request header*. Without this **trace_id** we could not create an APM span.

We modified dd-trace-cpp library *to allow late assignment of* **trace_id**.
Our C++ services are fully observable now.

Putting It All Together



Common AWS Inefficiencies

- Cold starts — Lambda/container init adds 1-3s silently
- Chatty APIs — 100 individual DynamoDB calls vs. one BatchGet
- Oversized payloads — full objects serialized, 3 fields actually used
- Retry storms — no jitter on backoff, failures amplify themselves
- Wrong instance type — memory-bound workload on compute-optimized node
- Missing connection pooling — new HTTPS connection per internal API call

In most cases:

- Visible in Datadog traces
- Reproducible under k6 load
- Root cause verified with Catch2

Key Takeaways

1. Measure at every layer — code, system, production
2. Catch2 BENCHMARK is underused in C++ shops — start now
3. The “mean” lies — always set SLOs on p99
4. Instrument your C++ code explicitly for Datadog
5. When layers disagree, that is where the bug is!

Key Takeaways

1. Measure at every layer — code, system, production

   Each layer catches what the others miss

2. Catch2 BENCHMARK is underused in C++ shops — start now

   Prevent regressions before they reach users

3. The “mean” lies — always set SLOs on p99

   Tail latency is where user pain lives

4. Instrument your C++ code explicitly for Datadog

   Observability does not happen by itself

5. When layers disagree, that is where the bug is!

Thank You

Questions?

[https://linkedin.com/in/alexcohn]