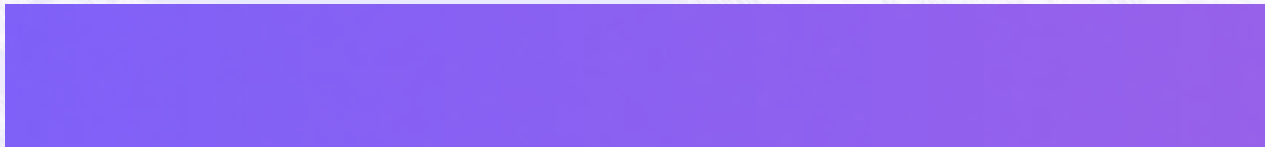


Making OTTL Go Brrr



How we achieved a 40x speedup with custom codegen for OTTL

Amir Jakoby / Sawmills.ai

**"A customer called.
Observability was degrading."**

No crashes. No spikes. Just... silence.



Agenda

01



What is Sawmills?

AI-powered telemetry management. We sit between your infrastructure and your observability tools, optimizing data in-flight. Built on the OpenTelemetry Collector.



OpenTelemetry & The Collector

OPENTELEMETRY

The open-source observability standard — vendor-neutral APIs, SDKs, and tools for **traces, metrics, and logs**. Second most active CNCF project after Kubernetes.

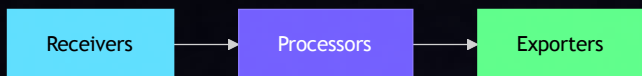
WHAT ARE PROCESSORS?

The middle layer — **filter, transform, sample, enrich** — before export.

THE COLLECTOR

A **single Go binary** that forms the telemetry pipeline. Pluggable architecture — you compose it from components:

Processors run on every record. Performance = throughput.



The Sawmills Constraint

Sawmills runs **inside the customer's infrastructure**. Our value proposition is cutting costs — so we can't be expensive ourselves.

- ◆ **Limited CPU** — processing all customer logs
must stay cheap
- ◆ **Limited memory** — 2-4 GiB to queue and
process TBs of data
- ◆ **Every allocation matters** — GC runs in our
memory budget



The Incident

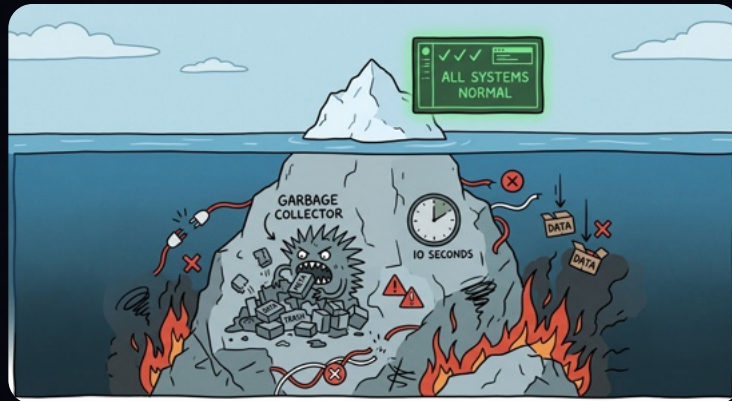
When observability goes quiet



The Customer

Every log record passes through **hundreds of OTTL expressions** before reaching its destination. The pipeline had been humming along for months.

We thought everything was fine. We were very, very wrong.



The Symptom

"We're seeing gaps in our observability data. Logs are disappearing, but nothing in the pipeline is reporting errors."

The worst kind of failure: the system that fails silently.



The Blind Spot

WHAT WE HAD

- ◆ Collector health metrics (up/down)
- ◆ Exporter success/failure counts
- ◆ Receiver metrics — **all green**, no errors
- ◆ Pipeline throughput at the output

WHAT WE WERE MISSING

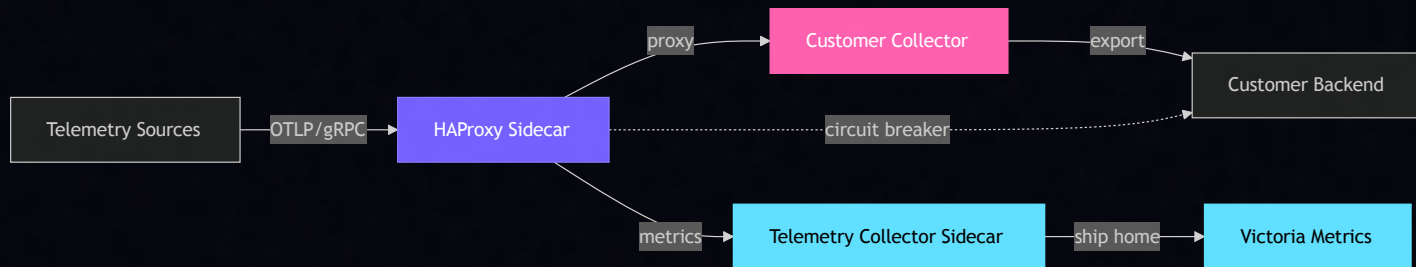
- ◆ **Pre-ingest** request rates
- ◆ Request latency **before** the Collector
- ◆ Rejected connections the receiver never saw
- ◆ What was being dropped **before** it reached the Collector

Key insight: If you can't see what's coming in, you can't tell what's being lost.



HAProxy Sidecar + Telemetry Collector

We introduced an HAProxy sidecar for **pre-ingest visibility**, with metrics shipped home via a separate telemetry collector.



What HAProxy Revealed

THE HIDDEN REJECTION

The Collector was **too busy evaluating OTTL expressions** to accept new connections. No goroutine available
→ gRPC rejected at the socket → **no error logged** because the Collector never saw the request.

Without HAProxy, these rejections were **completely invisible**. The Collector can't report what it never received.



The Investigation

Going deeper



Profiling the Collector

CPU profiling showed **OTTL expression evaluation** dominating the hot paths.

```
// Simplified flame graph - width = CPU time
```

```
main.processLogs 100%
```

```
  filterSampling.processLogs 89%
```

72% of CPU in OTTL evaluation. **malloggc** (Go allocator) at 24%.



The Allocation Problem

Every OTTL evaluation allocates. **300+ filters x thousands of records/sec** = millions of allocations/sec.

WHERE ALLOCATIONS HAPPEN

```
// Path resolution returns interface{}  
func resolvePath(path Path) any { ... } // alloc!  
  
// Values boxed to interface{}  
func compareValues(a, b any) bool { ... } // alloc!  
  
// Map lookups return interface{}  
val, _ := attrs.Get("key") // alloc!
```

THE MATH



Why It Matters

AST + Allocations



GC Pressure



Throughput Collapse



Silent Drops

The Vicious Cycle



Understanding OTTL

The OpenTelemetry Transformation Language



What is OTTL?

A DSL FOR TELEMETRY

OTTL is a domain-specific language built into the OpenTelemetry Collector for **filtering and transforming** telemetry data.

Used in processors like:

- ◆ `filter` — drop records by condition
- ◆ `transform` — modify attributes, body, severity
- ◆ `routing` — send data to different exporters

WHAT MAKES IT POWERFUL

- ◆ Declarative expressions in YAML config
- ◆ Access to all signal fields (attributes, body, resource)
- ◆ Built-in functions (IsMap, Contains, IsMatch, ExtractGrokPatterns, etc.)
- ◆ No custom code needed for common operations

The tradeoff: **expressiveness at the cost of runtime interpretation.**



The Real Data

This is what each OTTL filter runs against — thousands of times per second

Log Record (payment-service)

```
{
  "body": {
    "message": "Payment processed successfully",
    "transaction_id": "txn-12345-abcde",
    "amount": 9999,
    "metadata": {
      "card_type": "visa",
```



OTTL: Simple to Complex



300+ filters × 10K records/sec — each one walks the AST, boxes values, allocates memory



How OTTL Evaluates (Default)

Expression

parse →

AST

walk →

Resolve Paths

alloc →

Box Values

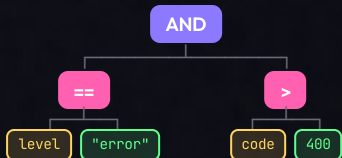
alloc →

Compare

→

bool

```
attrs["level"] == "error" and attrs["code"] > 400
```



300 filters = 300 trees walked per record.



"OTTL expressions are static. The data changes, the expressions don't."

We can compile them.



Approach 1

Code Generation



The Idea — CodeGen

What if we **generate native Go code** from OTTL expressions, compile it, and load it at startup?

Compile to `.so` plugin → `plugin.Open()` at startup → native function pointers at runtime.



The Pipeline

OTTL Expression

parse →

AST

visit →

Emit Go Code (jen)

build →

.so Plugin

open →

Native FilterFunc

```
// gen/generator.go - Parse OTTL, visit AST, emit Go code
func (g *Generator) Parse(expr string, mainFuncName string) ([]jen.Code, error) {
    ast, err := otll.ParseCondition(expr)
    // ...
    v := visitor.New(g.emitDebug, mainFuncName)
    codes, err := v.BooleanExpression(visitor.NewLocalStack(g.logger, "root"), ast)
    return codes, nil
}
```

The [jen](#) library generates type-safe Go code from the AST. No string templates, no code injection risk.



Generated Code Example

`attributes["level"] == "error" and attributes["status_code"] > 400` compiles to:

OTTL (INTERPRETED)

12 steps

GENERATED GO

0 allocs

1. Walk AST → BooleanExpr
2. Walk left → Comparison
3. Resolve "attributes" path
4. Lookup "level" key → any ← alloc
5. Box string → interface{} ← alloc



CodeGen vs OTTL — CPU Profile

What happens when you eliminate the interpreter entirely.

OTTL Interpreter 57.7 ns • 2 allocs

CodeGen Plugin 8.1 ns • 0 allocs

7x faster. Zero overhead — compiled to native Go, loaded as a plugin. But at what cost? →



Trade-offs — CodeGen

CodeGen gave us **3-12x speedups** and **near-zero allocations**. Tempting. But shipping it meant living with:

THE PROBLEM

fragile

WHAT WE NEED

portable

CodeGen is fast but fragile. We needed the same zero-alloc gains without the operational complexity.
Enter the closure combinator →



Approach 2

Closure Combinator



The Idea — Closure Combinator

Compile OTTL expressions into **chains of Go closures** at startup. No AST walking at runtime.

BEFORE (INTERPRETER)

per record

AFTER (CLOSURE)

0 allocs

× 300 filters × 10K records/sec

5 steps → 1 step. All the heavy lifting moves to startup. The hot path is a single function call.



What is a Closure Combinator?

CLOSURES IN GO

A function that **captures variables** from its scope. No lookups at call time.

```
key := "level" // captured once
getter := func(lr plog.LogRecord) string {
    val, _ := lr.Attributes().Get(key)
    return val.Str()
}
```

COMBINATOR PATTERN

Compose small closures into larger ones. Each OTTL operation → a closure:

All decisions made at compile time. At runtime, it's just function calls — no AST, no type assertions, no allocations.



Type Specialization — Kill the Boxing

The core trick: replace **interface** with **typed getters**. No boxing = no allocations.

GENERIC (DEFAULT) allocates

```
type valueGetter func(ctx, lr, resource, scope) any // ← bo
```

TYPE-SPECIFIC 0 allocs

```
type stringGetter func(ctx, lr, resource, scope) (string, b  
type int64Getter func(ctx, lr, resource, scope) (int64, bo
```

Comparisons become type-specific — **no boxing, no type assertion**:

```
func makeStringComparison(left stringGetter, right string, op CompareOp) LogFilterFunc {  
    return func(ctx, lr, resource, scope) bool {  
        l, ok := left(ctx, lr, resource, scope)  
        return ok && l == right // direct comparison, zero allocs  
    }  
}
```

Compile-time type knowledge eliminates runtime boxing. The Go compiler sees concrete types and optimizes accordingly.



Three-Tier Compilation

At compile time, try the **fastest path first**. Only fall back when the type doesn't match.

```
func (c *LogCompiler) compileComparison(comp *ottl.Comparison) (LogFilterFunc, error) {  
    // Tier 1: string (most common, zero allocs)  
    if fn := c.tryStringComparison(comp); fn != nil { return fn, nil }  
  
    // Tier 2: int64 (zero allocs)  
    if fn := c.tryInt64Comparison(comp); fn != nil { return fn, nil }  
  
    // Tier 3: generic fallback (interface boxing)  
    leftGetter, _ := c.compileValueGetter(&comp.Left)  
    rightGetter, _ := c.compileValueGetter(&comp.Right)  
    return makeComparison(leftGetter, rightGetter, comp.Op), nil  
}
```

90%+ of comparisons hit Tier 1 or 2. Each resolves to a specialized closure (`stringEq`, `int64Gt`, `int64Lt`, ...) — at runtime, it's already a direct typed comparison. Zero branching, zero allocs.



Zero-Alloc Path Resolution

Paths like `attributes["key"]` are resolved **once at startup**, not on every record.

BEFORE

4 steps per eval

AFTER

1 call

Same pattern for **resource attributes**, **body paths**, and **nested map access** — every path becomes a precompiled closure.



The Proof — Real CPU Profile

After applying closure compilation, the flame graph transforms completely.

Before

255.7 ns · 4 allocs

After

31.2 ns · 0 allocs

27% of CPU wasted on overhead → 0%. Every cycle now does actual work.



Results

The numbers



40x

From bottleneck to headroom

8-12x

per expression

×

300+

filters

×

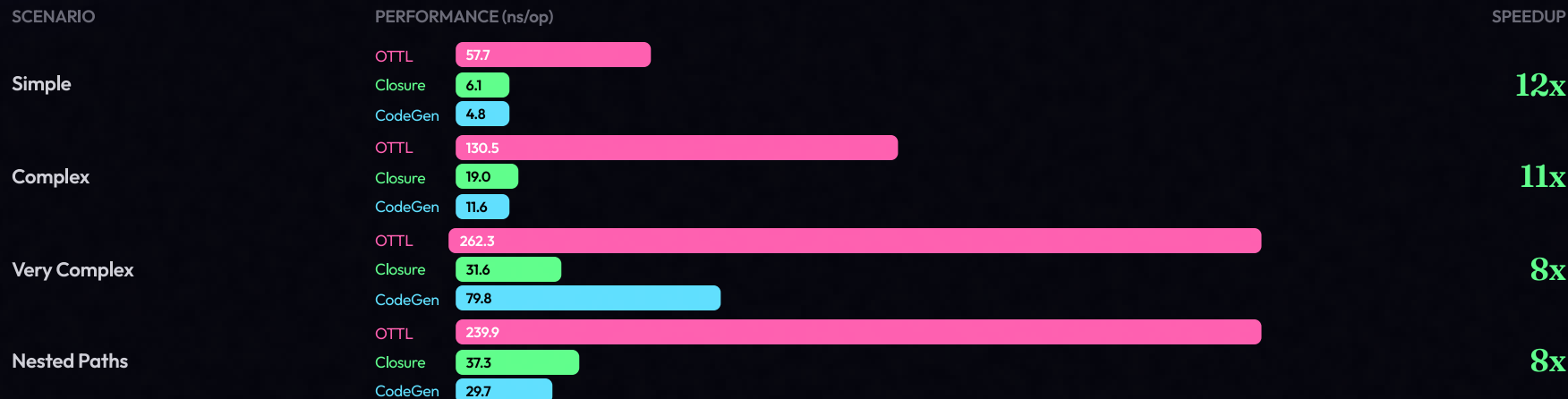
0

GC pressure

No code changes for customers. No infrastructure changes. Just a smarter engine.



Benchmark Breakdown



Closure: 0 allocs/op on all paths. GC pressure eliminated.

Regex: 1.6x — regex engine dominates, not our code. Can we fix that? →



What About Regex? — The Last Frontier

Go's regexp uses Thompson NFA — safe, but no hardware acceleration

Intel — Hyperscan

- ◆ AVX2 / AVX-512 SIMD acceleration

ARM — Vectorscan

The play: compile all OTTL regex patterns into one Hyperscan/Vectorscan database → scan once per record → match all patterns in one pass

~1.6x → 50x+
potential regex speedup



What We Shipped

In Production DEFAULT

On the Shelf INTERNAL

Closure Combinator 0 allocs

- ◆ **Default engine** for all customers
- ◆ Pure Go — portable, no CGO
- ◆ Zero allocations on all evaluated paths

Best trade-off: 90% of the speedup, 0% of the constraints



Lessons Learned

What we'd tell ourselves before this started

1 Measure First



Questions? — Let's Talk

Making OTTL Go Brrr

We're building SIMD-accelerated regex for OTTL — open source

Interested in contributing or testing? Let's chat after — or connect online



Amir Jakoby

[Sawmills.ai](#) — Smart telemetry management

linkedin.com/in/amirjakoby

Scan to reach out