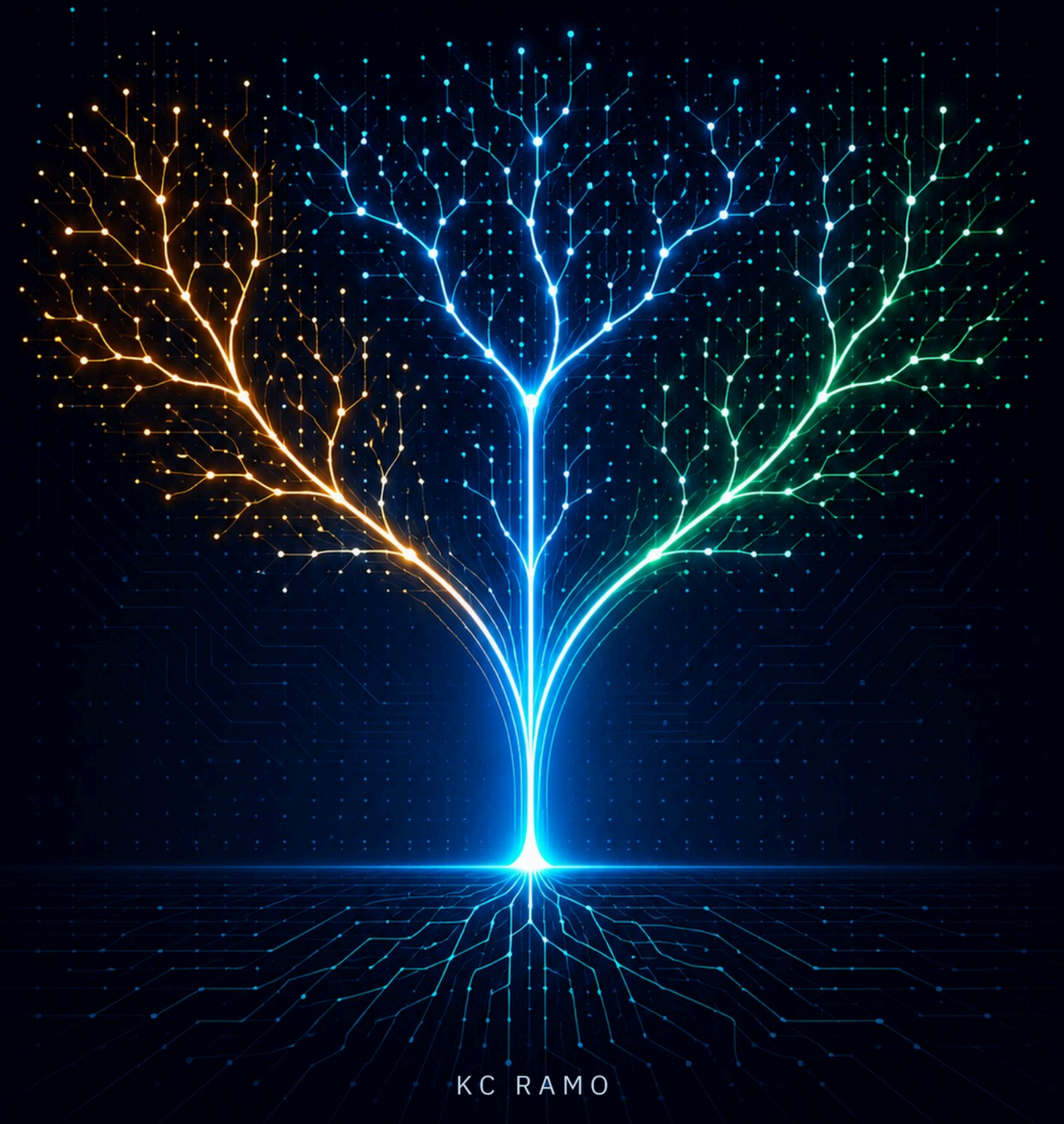
THE SIGNAL PATH

Build the Tools That Watch Everything Else



KC RAMO

TECHNOVIZE PUBLISHING

The Signal Path

Build the Tools That Watch Everything Else

KC Ramo

Technovize Publishing · 2026
Amsterdam, The Netherlands

The Signal Path · First edition

Published by **Technovize Publishing**
Amsterdam, The Netherlands

Colophon — This book was written in Markdown. Code samples were tested against Go 1.22, Python 3.12, and Rust 1.78 on Ubuntu 24.04 LTS. Set in IBM Plex Serif, Sans & Mono. Rendered with WeasyPrint. Printing 1 · 2026-07.

To the engineers who debugged a production outage with `print()` statements at 3 a.m. There is a better way. Let's build it.

About This Book

Every production incident begins the same way: a dashboard turns red, a phone buzzes, and someone asks, "What changed?" The quality of the answer — and the speed at which it arrives — depends entirely on the observability platform underneath. Yet most engineers only use observability tools. They configure Grafana panels. They write PromQL queries. They copy-paste OpenTelemetry snippets. They never see the internals.

This book changes that.

The Signal Path follows **Lens**, an observability platform built entirely from scratch. Starting with a single agent that reads raw Linux `/proc` files, we will build: a time-series database, a structured logging pipeline, a distributed tracing engine, an anomaly detection system, and a global-scale observability data lake. Every component is built from first principles before we compare it to the industry standard (Prometheus, Loki, OpenTelemetry, ClickHouse) that does the same thing.

By the end, you will not just use observability tools. You will understand them at the level of the engineers who built them.

Preface

Why Observability Needs a Build-Along Book

There are two kinds of technical books. The first kind teaches you how to use a tool: *"Click this button to create a Grafana dashboard. Write this PromQL query to get CPU utilization."* These books are useful for about six months — until the tool's UI changes.

The second kind teaches you how the tool works underneath: *"Here is how Prometheus compresses time-series data. Here is the ring buffer that makes it fast. Here is why the TSDB stores data in two-hour blocks."* These books remain useful for decades.

This book is the second kind.

Every chapter follows the same pattern: **build the toy version first, understand the principles, then compare to the production tool.** The toy version is always under 500 lines of code. The principles are extracted and numbered. The production comparison shows you exactly what the real tool adds.

By the end of this book, you will have written approximately 5,000 lines of Go, 3,000 lines of Python, and 1,000 lines of Rust across 16 chapters. You will have a working observability platform — not production-grade, but production-shaped. You will understand why Prometheus uses a TSDB instead of a relational database. Why Loki indexes logs by label, not by content. Why OpenTelemetry exists as a standard. Why Datadog costs $15 per host per month and whether that is a good deal for your organization.

Most importantly, you will stop being a user of observability tools and start being a builder of observability systems. Users ask "how do I configure this?" Builders ask "how does this work?" One question has a six-month shelf life. The other lasts a career.

Welcome to the second kind of book. Let's build something.

Who Should Read This Book

- **Backend and infrastructure engineers** who want to understand observability internals beyond the dashboard.
- **SREs and platform engineers** who build monitoring for their organizations and need to make trade-off decisions.
- **Senior engineers** preparing for system design interviews where observability architecture is part of the discussion.
- **Curious engineers** who have configured one too many YAML files and want to know what is actually happening underneath.

What You Need to Know

You should be comfortable reading at least one of Go, Python, or Rust — the book uses all three, but each language is introduced and explained at the point of use. You should have deployed at least one application to production and wondered why it was slow. No prior experience with Prometheus, Grafana, OpenTelemetry, or any observability tool is assumed.

How to Use This Book

Read it in order. Each chapter builds on the Lens codebase as it stood at the end of the previous chapter. The complete source code for every chapter is available at the companion repository: github.com/DjangoZenDev/signal-path.

Contents

FRONT MATTER

About This Book iv

Preface v

Abbreviations xv

PART I · FOUNDATION — COLLECTING SIGNALS

1 The Single Agent 2

1.1 The Outage That Started Everything 3

1.2 What We Are Building 4

1.3 Why Go? 4

1.4 The Linux /proc Filesystem 5

1.5 The Lens Agent: Core Architecture 6

1.6 The CPU Collector 9

1.7 The Memory Collector 11

1.8 The File Descriptor Collector 13

1.9 The HTTP Server 14

1.10 Running Lens in Production 17

1.11 What We Built 17

1.12 The Principles So Far 18

1.13 What's Next 19

2 The First Thousand Metrics 20

2.1 The Problem That Emerged Naturally 21

2.2 The Time-Series Data Model 22

2.3 The Ring Buffer: In-Memory Storage 24
2.4 The Gorilla Compression Algorithm 27
2.5 The Write-Ahead Log 30
2.6 Putting It Together: The TSDB Core 32
2.7 The Prometheus TSDB 36
2.8 Trade-off Analysis 38
2.9 Principles from Chapter 2 39
2.10 What's Next 40
3 Logs Are Just Events 41
3.1 The Night of One Thousand Greps 42
3.2 The Problem with `print()` 43
3.3 What We Are Building 44
3.4 The Lens Logger: Design 45
3.5 The Log Event Schema 46
3.6 The Ring Buffer Implementation 48
3.7 The Segment Writer 50
3.8 The Inverted Index 52
3.9 The HTTP Ingestion API 53
3.10 Trade-Off Analysis 55
3.11 Real-World Comparison: Grafana Loki 56
3.12 The Principles So Far 57
3.13 What's Next 58
4 The Agent Grows 59
4.1 The Bug That Could Not Be Logged 60
4.2 What eBPF Actually Is 61

4.3 Our First eBPF Program: Counting Syscalls 62
4.4 Building eBPF Programs 67
4.5 Integrating eBPF into Lens 68
4.6 Beyond Syscalls: A Complete eBPF Suite 70
4.7 The Lens Agent Architecture, Chapter 4 Edition 72
4.8 Performance: Can eBPF Handle Production Load? 74
4.9 Security: What Can't Lens See? 75
4.10 Trade-off: eBPF vs. Sidecar Agents vs. Application SDKs 75
4.11 Principles from Chapter 4 76
4.12 What's Next 77

PART II · CORRELATION — CONNECTING SIGNALS

5 Traces and the Span 80
5.1 The Latency That Had No Owner 81
5.2 The Span Data Model 81
5.3 W3C TraceContext Propagation 84
5.4 Building a Minimal Tracer 87
5.5 Head-Based vs Tail-Based Sampling 90
5.6 The Cost of Tracing 92
5.7 Comparing to OTLP 93
5.8 Principles from Chapter 5 94
5.9 What's Next 95
6 The Unified Pipeline 97
6.1 The Fragmentation Problem 98
6.2 The OTel Collector Architecture 99
6.3 The Pipeline as a Directed Graph 100

6.4 Building a Custom Filtering Processor 104
6.5 OTLP: The Protocol That Makes This Work 107
6.6 Writing Spans to SQLite 108
6.7 Hot-Reload: Watching Configuration 111
6.8 Performance and Trade-offs 113
6.9 Principles from Chapter 6 113
6.10 What's Next 114
7 The Three Pillars 115
7.1 The Incident That Proved the Point 116
7.2 Exemplars: The Bridge Between Metrics and Traces 117
7.3 Building the Correlation Engine 120
7.4 The Golden Signal Dashboard 124
7.5 Debugging a Real Incident with All Three Pillars 126
7.6 Trade-offs in Correlation Design 127
7.7 Principles from Chapter 7 128
7.8 What's Next 129
8 Service Maps and Topology 131
8.1 The Architecture Diagram That Was A Lie 132
8.2 The Service Graph Data Model 133
8.3 Building the Graph from the Span Pipeline 136
8.4 PageRank for Critical Services 138
8.5 D3.js Force-Directed Visualization 141
8.6 Anomaly Detection in the Graph 144
8.7 Comparison: Jaeger / Grafana Node Graph 146
8.8 Principles from Chapter 8 148

8.9 What's Next 149

PART III · INTELLIGENCE — UNDERSTANDING SIGNALS

9 Anomaly Detection 151

9.1 The Alert That Cried Wolf 152

9.2 The Problem with Static Thresholds 153

9.3 Approach 1: Moving Average with Dynamic Bands 153

9.4 Approach 2: Exponential Smoothing (Holt-Winters) 155

9.5 Approach 3: STL Decomposition 159

9.6 Approach 4: Isolation Forests for Multivariate Detection 162

9.7 Putting It Together: The Lens Anomaly Engine 166

9.8 Comparison: The State of the Art 168

9.9 The False Positive Problem 170

9.10 Principles from Chapter 9 171

9.11 What's Next 172

10 Root Cause Analysis 173

10.1 The Incident That Broke the Pattern 174

10.2 The Causal Graph Data Model 175

10.3 Building the Dependency Graph from Traces 176

10.4 Differential Diagnosis: Tracing Latency Backward 178

10.5 The "Five Whys" of Distributed Systems 181

10.6 The Blame Attribution Score 182

10.7 Implementing in Go: The Causal Engine 183

10.8 Comparison: Google Dapper, AWS X-Ray, Jaeger 186

10.9 When the Algorithm Fails 188

10.10 Principles from Chapter 10 189

10.11 What's Next 190

11 Predictive Observability 191

11.1 The Disk That Failed at 3 AM 192

11.2 The Simple Thing: Linear Trend Extrapolation 193

11.3 When Linear Fails: Seasonality and Prophet 195

11.4 When Both Fail: LSTM for Complex Patterns 197

11.5 The Cost Calculus: False Positives vs. False Negatives 200

11.6 Comparison: Real-World Predictive Tools 202

11.7 When Prediction Is Not Worth It 203

11.8 Principles from Chapter 11 204

11.9 What's Next 205

12 The Dashboard That Thinks 206

12.1 The Dashboard Graveyard 207

12.2 SLOs as Code 208

12.3 Computing Error Budgets in Real Time 210

12.4 The Dashboard That Highlights Change 212

12.5 Auto-Generating Dashboards from Service Metadata 214

12.6 Build: The Change-Detection Dashboard Engine 216

12.7 Comparison: Grafana SLO, Datadog SLO, Google SRE Workbook 221

12.8 Principles from Chapter 12 223

12.9 What's Next 224

PART IV · SCALE — OPERATING SIGNALS

13 The Observability Data Lake 226

13.1 The $47,000 Surprise 227

13.2 The Three Tiers 227

13.3 Building the Retention Policy Engine 228
13.4 Downsampling: Trading Resolution for Cost 230
13.5 Query Federation 231
13.6 Cost Modeling: What 1PB Costs 232
13.7 Compaction and the Write Amplification Problem 233
13.8 Schema Evolution Across Tiers 234
13.9 Real-World Comparisons 234
13.10 Principles 235
13.11 Operational Realities of the Data Lake 236
13.12 What's Next 238
14 Multi-Cluster, Multi-Cloud 239
14.1 The Federation Problem 240
14.2 Deduplication 241
14.3 Latency Budgets Across Oceans 241
14.4 Handling Partial Failures 242
14.5 Multi-Cloud Authentication and Networking 243
14.6 Global Rate Limiting 243
14.7 Multi-Cluster Alerting 244
14.8 Case Study: The European Expansion 244
14.9 Federated SLO Tracking 245
14.10 Disaster Recovery: When a Region Disappears 246
14.11 Principles 247
14.12 What's Next 247
15 The Cost of Seeing 248
15.1 The $47,000 Wake-Up Call 249

15.2 Cardinality: The Silent Budget Killer 249
15.3 Sampling by Cost 249
15.4 Build vs. Buy 250
15.5 The Build Decision Framework 250
15.6 Cost Attribution 251
15.7 The Psychology of Spending 251
15.8 Real Numbers at Scale (500 Nodes) 251
15.9 Hidden Costs 252
15.10 When to Stop Watching 253
15.11 Principles 253
16 The Principles That Remain 255
16.1 The 22 Principles 256
16.2 Anti-Patterns: The Five Ways Observability Programs Fail 257
16.3 The Architecture Decision Framework 257
16.4 Dara's Final Reflection 258
16.5 The Signal Path Forward 258
APPENDICES
A Signal Taxonomy 261
B Tool Comparison Matrix 265
C Further Reading 268
D Glossary of Observability Terms 271
E Companion Repository Guide 274
F Tools and Languages Quick Reference 278
About the Author 284

Abbreviations

Abbreviation	Stands For	Chapter
API	Application Programming Interface	1
ASGI	Asynchronous Server Gateway Interface	9
BPF / eBPF	Extended Berkeley Packet Filter	4
CAP	Consistency, Availability, Partition Tolerance (theorem)	14
CDC	Change Data Capture	13
CRDT	Conflict-Free Replicated Data Type	9
CQRS	Command Query Responsibility Segregation	12
DLQ	Dead Letter Queue	8
GDPR	General Data Protection Regulation	14
gRPC	gRPC Remote Procedure Call	5
IOPS	Input/Output Operations Per Second	6
LGTM	Loki, Grafana, Tempo, Mimir (Grafana stack)	14
LSTM	Long Short-Term Memory (neural network)	11
mTLS	Mutual Transport Layer Security	14
MVCC	Multi-Version Concurrency Control	2
NTP	Network Time Protocol	14
NVMe	Non-Volatile Memory Express (fast SSD)	13

Abbreviation	Stands For	Chapter
OLAP	Online Analytical Processing	12
OLTP	Online Transaction Processing	12
OTel	OpenTelemetry	5
OTLP	OpenTelemetry Protocol	5
P50/P95/P99	50th/95th/99th Percentile (latency)	1
PCI	Payment Card Industry (security standard)	13
PID	Process Identifier	1
PTP	Precision Time Protocol	14
QPS	Queries Per Second	2
RED	Rate, Errors, Duration (method)	12
RTT	Round-Trip Time	14
SLA	Service Level Agreement	12
SLI	Service Level Indicator	12
SLO	Service Level Objective	12
SRE	Site Reliability Engineering	1
SSD	Solid-State Drive	13
STL	Seasonal-Trend decomposition using Loess	9
TSDB	Time-Series Database	2
USE	Utilization, Saturation, Errors (method)	1

Abbreviation	Stands For	Chapter
WAL	Write-Ahead Log	2
WSGI	Web Server Gateway Interface	1

I

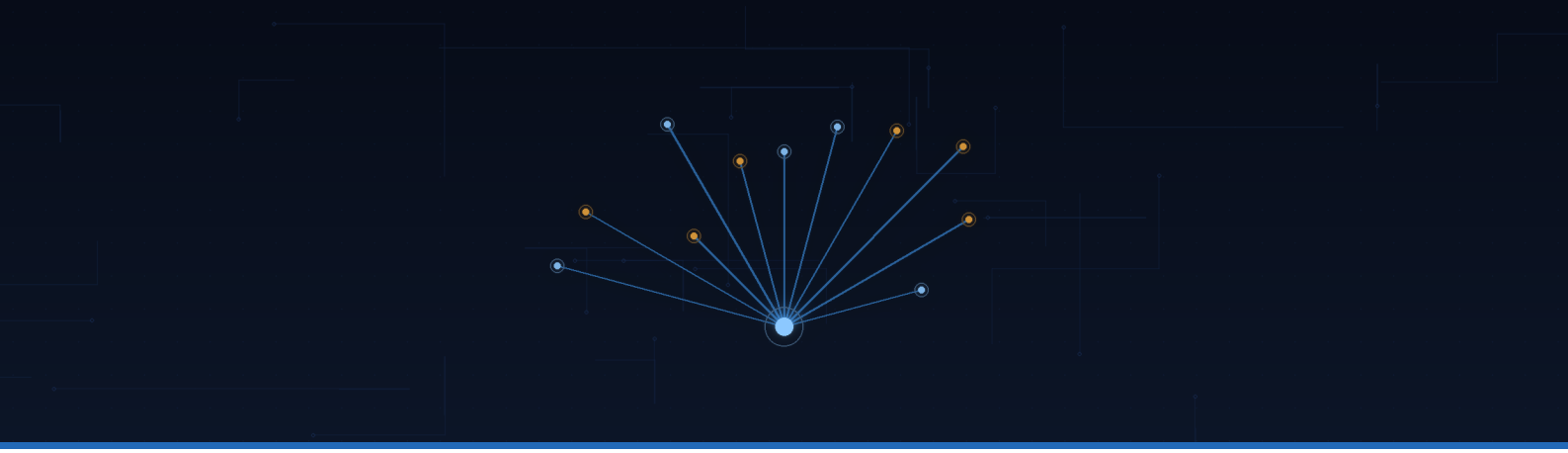
PART ONE

Foundation — Collecting Signals

Every observability platform begins with a single binary on a single machine, asking the kernel what is happening right now. This part builds that binary — the Lens agent — and teaches it to store metrics, aggregate logs, and see inside the kernel with eBPF.

CHAPTERS IN THIS PART

1	The Single Agent
2	The First Thousand Metrics
3	Logs Are Just Events
4	The Agent Grows



CHAPTER 01

Part I — Foundation

The Single Agent

A Go binary that reads /proc. CPU, memory, disk, network.
The simplest thing that produces a metric.

IN THIS CHAPTER

1.1 The Outage That Started Everything

1.2 What We Are Building

1.3 Why Go?

1.4 The Linux /proc Filesystem

1.5 The Lens Agent: Core Architecture

1.6 The CPU Collector

> *Every observability platform begins with a question: what is happening right now? The answer always comes from the same place — the kernel. Everything else is just packaging.*

1.1 The Outage That Started Everything

Arjun Mehta was the second SRE at OrbitPay, a fintech startup processing $200 million in transactions per day. On a Tuesday in March, the payment processor went down for 47 minutes. Customers saw "Transaction Failed." Merchants saw empty dashboards. The CEO saw a $1.8 million revenue hole.

The root cause, when Arjun finally found it at 4:23 AM, was a file descriptor leak in a microservice nobody had updated in six months. The service had been quietly consuming file handles for weeks. When it hit the kernel limit of 1,024, every new connection failed. The service itself was still running — CPU at 2%, memory at 15%, health check passing. By every standard metric, it looked healthy. It was not.

The postmortem revealed a damning fact: OrbitPay's observability stack consisted of CPU, memory, and disk graphs from a default CloudWatch dashboard. There was no file descriptor monitoring. No connection pool monitoring. No application-level metric that would have surfaced "open file handles approaching limit" as a flashing red number.

Arjun spent the next week building what he wished he'd had: an agent that could read any kernel metric, expose it in a standard format, and alert on thresholds. He called it **Lens**.

This chapter is about building Lens v0.1 — a single Go binary that reads the `/proc` filesystem, extracts useful metrics, and exposes them over HTTP. It is the simplest possible observability platform. It is also the foundation for everything that follows.

1.2 What We Are Building

Before we write a line of Go, let us be precise about what Lens v0.1 must do:

Capability	Requirement
Read kernel metrics	CPU usage, memory usage, disk I/O, network bytes, open file descriptors, process count
Expose metrics	HTTP endpoint at `/metrics` returning metrics in a structured text format
Low overhead	Agent must consume less than 1% CPU and less than 50 MB of memory
No dependencies	Single static binary. No runtime. No package manager required.
Run anywhere	Linux 4.15+ kernel. If `/proc` exists, Lens works.

That is the entire specification. Notice what is not here: no Prometheus compatibility (Chapter 2), no log aggregation (Chapter 3), no eBPF probes (Chapter 4), no tracing (Chapter 5), no anomaly detection (Chapter 9). Each piece arrives when the system demands it, not before.

1.3 Why Go?

Lens is written in Go for three reasons:

Single static binary. Go compiles to a single executable with zero dependencies. Deploying Lens to 10,000 servers means copying one file. No Python virtual environment. No Node.js `node_modules`. No JVM runtime. This matters because observability agents run *everywhere* — including minimal container images and embedded systems where a Python interpreter is 100 MB of overhead.

Concurrency is free. Go's goroutine model makes it trivial to collect multiple metrics simultaneously without complex threading code. Each `/proc` file can be read in its own goroutine, results collected over a channel, and the whole thing converges in microseconds.

Performance without tuning. Go's garbage collector is optimized for the kind of short-lived allocations that metric collection produces. A well-written Go agent can run at < 0.5% CPU on a production machine without any tuning. Python, by contrast, typically needs careful attention to avoid CPU spikes during garbage collection.

But the concepts in this book are language-agnostic. Every algorithm is explained in pseudocode first, with the Go implementation following. If you work in Rust, C, or Zig, you will be able to port every chapter.

1.4 The Linux /proc Filesystem

Every metric Lens collects comes from `/proc`, a virtual filesystem maintained by the Linux kernel. Unlike a real filesystem stored on disk, `/proc` is generated on the fly — when you read `/proc/cpuinfo`, the kernel computes the current CPU state and returns it as text. There are no disk I/O costs. Reads are effectively memory-speed.

The key files for Lens v0.1:

File	Contains	Metrics extracted
`/proc/stat`	CPU time accounting	`cpu.user`, `cpu.system`, `cpu.idle`, `cpu.iowait`
`/proc/meminfo`	Memory statistics	`memory.total`, `memory.available`, `memory.used`, `memory.cached`
`/proc/diskstats`	Disk I/O per device	`disk.read_bytes`, `disk.write_bytes`, `disk.read_ops`, `disk.write_ops`
`/proc/net/dev`	Network per interface	`net.rx_bytes`, `net.tx_bytes`, `net.rx_packets`, `net.tx_packets`

File	Contains	Metrics extracted
`/proc/loadavg`	System load averages	`load.1min`, `load.5min`, `load.15min`
`/proc/sys/fs/file-nr`	Open file handles	`files.open`, `files.max`
`/proc/<pid>/stat`	Per-process statistics	`process.cpu`, `process.memory`, `process.fds`

The format of these files is not JSON. It is not YAML. It is not even consistent. `/proc/stat` uses space-separated columns. `/proc/meminfo` uses `Key: value` pairs with units like `kB`. `/proc/diskstats` uses 20 columns of whitespace-separated integers. Parsing them is a lesson in the organic evolution of kernel interfaces — and a reminder that the most important interfaces in computing are often the ugliest.

1.5 The Lens Agent: Core Architecture

agent.go GO

```go
// agent.go — Lens v0.1: The single agent
package main

import (
    "fmt"
    "net/http"
    "sync"
    "time"
)

// Metric represents a single observation at a point in time.
type Metric struct {
    Name   string
    Value  float64
    Labels map[string]string
```

```
    Time   time.Time
}

// Collector is the interface that every metric source implements.
type Collector interface {
    Name() string
    Collect() ([]Metric, error)
}

// Agent orchestrates all collectors and exposes metrics over HTTP.
type Agent struct {
    collectors []Collector
    metrics    []Metric
    mu         sync.RWMutex
}

func NewAgent() *Agent {
    return &Agent{
        collectors: []Collector{
            &CPUCollector{},
            &MemoryCollector{},
            &DiskCollector{},
            &NetworkCollector{},
            &LoadCollector{},
            &FileDescriptorCollector{},
        },
    }
}

// Run starts the collection loop in the background.
func (a *Agent) Run(interval time.Duration) {
    go func() {
        ticker := time.NewTicker(interval)
        for range ticker.C {
            a.collect()
        }
    }()
}
```

```
func (a *Agent) collect() {
    var allMetrics []Metric

    // Collect from each source concurrently.
    var wg sync.WaitGroup
    results := make(chan []Metric, len(a.collectors))

    for _, c := range a.collectors {
        wg.Add(1)
        go func(collector Collector) {
            defer wg.Done()
            metrics, err := collector.Collect()
            if err != nil {
                // In production, log the error. For v0.1, skip.
                return
            }
            results <- metrics
        }(c)
    }

    wg.Wait()
    close(results)

    for metrics := range results {
        allMetrics = append(allMetrics, metrics...)
    }

    a.mu.Lock()
    a.metrics = allMetrics
    a.mu.Unlock()
}
```

The agent has three responsibilities: collect, store, and expose. Collection happens in a background goroutine on a configurable interval (default: 10 seconds). Storage is in-memory — a simple slice of `Metric` structs protected by a read-write mutex. Exposure is an HTTP handler that serializes the current metrics into a text format.

This is intentionally minimal. No ring buffer (Chapter 2). No persistent storage (Chapter 2). No push to a remote server (Chapter 6). For v0.1, the agent is a stateless probe — it reads the kernel, holds the latest snapshot in memory, and answers HTTP requests.

1.6 The CPU Collector

The CPU collector is the simplest and most important collector in Lens. It reads `/proc/stat` and computes CPU utilization as a percentage:

collectors/cpu.go GO

```go
// collectors/cpu.go
package collectors

import (
    "bufio"
    "fmt"
    "os"
    "strconv"
    "strings"
)

type CPUCollector struct {
    prevIdle  uint64
    prevTotal uint64
}

func (c *CPUCollector) Name() string { return "cpu" }

func (c *CPUCollector) Collect() ([]Metric, error) {
    file, err := os.Open("/proc/stat")
    if err != nil {
        return nil, fmt.Errorf("open /proc/stat: %w", err)
    }
    defer file.Close()

```

```
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        line := scanner.Text()
        if !strings.HasPrefix(line, "cpu ") {
            continue
        }

        // cpu  user nice system idle iowait irq softirq steal guest guest_nice
        fields := strings.Fields(line)
        if len(fields) < 8 {
            continue
        }

        user, _ := strconv.ParseUint(fields[1], 10, 64)
        nice, _ := strconv.ParseUint(fields[2], 10, 64)
        system, _ := strconv.ParseUint(fields[3], 10, 64)
        idle, _ := strconv.ParseUint(fields[4], 10, 64)
        iowait, _ := strconv.ParseUint(fields[5], 10, 64)
        irq, _ := strconv.ParseUint(fields[6], 10, 64)
        softirq, _ := strconv.ParseUint(fields[7], 10, 64)
        steal, _ := strconv.ParseUint(fields[8], 10, 64)

        idleTotal := idle + iowait
        total := user + nice + system + idle + iowait + irq + softirq + steal

        // Compute delta since last collection (percentage).
        deltaIdle := idleTotal - c.prevIdle
        deltaTotal := total - c.prevTotal

        var cpuPercent float64
        if deltaTotal > 0 {
            cpuPercent = 100.0 * (1.0 - float64(deltaIdle)/float64(deltaTotal))
        }

        c.prevIdle = idleTotal
        c.prevTotal = total

```

```
63          now := time.Now()
64          return []Metric{
65              {Name: "cpu.utilization_percent", Value: cpuPercent,
    Time: now},
66              {Name: "cpu.user_seconds_total", Value:
    float64(user), Time: now},
67              {Name: "cpu.system_seconds_total", Value: float64(sys
    tem), Time: now},
68              {Name: "cpu.iowait_seconds_total", Value: float64(iow
    ait), Time: now},
69          }, nil
70      }
71
72      return nil, scanner.Err()
73  }
```

The key insight: `/proc/stat` provides cumulative counters since boot, not instantaneous values. To compute CPU utilization as a percentage, we must store the previous values and compute the delta on each collection. This pattern — cumulative counter → delta → rate — recurs through every metric backend in existence. Prometheus calls these "counter" metrics. The OpenTelemetry data model calls them "monotonic sums." Both are built on the same principle we implement here: subtract the previous value, divide by elapsed time.

1.7 The Memory Collector

Memory is simpler than CPU because `/proc/meminfo` provides absolute values:

collectors/memory.go GO

```
1   // collectors/memory.go
2   package collectors
3
4   import (
5       "bufio"
6       "fmt"
```

```
    "os"
    "strconv"
    "strings"
)

type MemoryCollector struct{}

func (c *MemoryCollector) Name() string { return "memory" }

func (c *MemoryCollector) Collect() ([]Metric, error) {
    file, err := os.Open("/proc/meminfo")
    if err != nil {
        return nil, fmt.Errorf("open /proc/meminfo: %w", err)
    }
    defer file.Close()

    values := make(map[string]float64)
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        fields := strings.Fields(scanner.Text())
        if len(fields) < 2 {
            continue
        }
        key := strings.TrimSuffix(fields[0], ":")
        val, _ := strconv.ParseFloat(fields[1], 64)
        // meminfo values are in kB. Convert to bytes.
        values[key] = val * 1024
    }

    now := time.Now()
    total := values["MemTotal"]
    available := values["MemAvailable"]
    used := total - available

    return []Metric{
        {Name: "memory.total_bytes", Value: total, Time: now},
        {Name: "memory.available_bytes", Value: available, Time:
now},
        {Name: "memory.used_bytes", Value: used, Time: now},
```

```
        {Name: "memory.used_percent", Value: (used / total) * 100.0, Time: now},
        {Name: "memory.cached_bytes", Value: values["Cached"], Time: now},
        {Name: "memory.swap_used_bytes", Value: values["SwapTotal"] - values["SwapFree"], Time: now},
    }, nil
}
```

1.8 The File Descriptor Collector

This is the collector Arjun needed during the outage. It reads the system-wide open file handle count and computes usage as a percentage of the kernel limit:

collectors/fd.go GO

```go
// collectors/fd.go
package collectors

import (
    "fmt"
    "os"
    "strconv"
    "strings"
)

type FileDescriptorCollector struct{}

func (c *FileDescriptorCollector) Name() string { return "file_descriptors" }

func (c *FileDescriptorCollector) Collect() ([]Metric, error) {
    data, err := os.ReadFile("/proc/sys/fs/file-nr")
    if err != nil {
        return nil, fmt.Errorf("read /proc/sys/fs/file-nr: %w", err)
    }
```

```
    // Format: "open_count  unused  max"
    // Example: "1536    0    9223372036854775807"
    fields := strings.Fields(string(data))
    if len(fields) < 3 {
        return nil, fmt.Errorf("unexpected /proc/sys/fs/file-nr format: %s", string(data))
    }

    open, _ := strconv.ParseFloat(fields[0], 64)
    max, _ := strconv.ParseFloat(fields[2], 64)

    now := time.Now()
    return []Metric{
        {Name: "files.open", Value: open, Time: now},
        {Name: "files.max", Value: max, Time: now},
        {Name: "files.used_percent", Value: (open / max) * 100.0, Time: now},
    }, nil
}
```

This is 30 lines of Go. During the OrbitPay outage, a collector like this — with an alert threshold at 80% — would have surfaced the file descriptor leak two weeks before it caused an outage. The cost of those 30 lines, measured in the $1.8 million that OrbitPay lost, was approximately $60,000 per line. This is the economics of observability in one number.

1.9 The HTTP Server

The agent exposes metrics over HTTP. This is the simplest possible interface — a plain-text format that a human can read and a machine can parse:

server.go GO

```
// server.go
package main
```

```
3
4   import (
5       "fmt"
6       "net/http"
7       "sort"
8   )
9
10  func (a *Agent) metricsHandler(w http.ResponseWriter, r *http.Req
    uest) {
11      a.mu.RLock()
12      defer a.mu.RUnlock()
13
14      w.Header().Set("Content-Type", "text/plain; version=0.0.4")
15
16      // Sort metrics by name for stable output.
17      sorted := make([]Metric, len(a.metrics))
18      copy(sorted, a.metrics)
19      sort.Slice(sorted, func(i, j int) bool {
20          return sorted[i].Name < sorted[j].Name
21      })
22
23      for _, m := range sorted {
24          // Format: metric_name{label="value"} value timestamp
25          labelStr := ""
26          if len(m.Labels) > 0 {
27              parts := make([]string, 0, len(m.Labels))
28              for k, v := range m.Labels {
29                  parts = append(parts, fmt.Sprintf("%s=%q", k, v))
30              }
31              sort.Strings(parts)
32              labelStr = "{" + strings.Join(parts, ",") + "}"
33          }
34
35          fmt.Fprintf(w, "%s%s %g %d\n",
36              m.Name, labelStr, m.Value, m.Time.UnixMilli())
37      }
38  }
39
40  func main() {
41      agent := NewAgent()
```

```
    agent.Run(10 * time.Second)

    http.HandleFunc("/metrics", agent.metricsHandler)
    http.HandleFunc("/health", func(w http.ResponseWriter, r *htt
p.Request) {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    })

    fmt.Println("Lens v0.1 listening on :9100")
    http.ListenAndServe(":9100", nil)
}
```

The output looks like:

```
cpu.iowait_seconds_total 1247.3 1744684800000
cpu.system_seconds_total 89134.1 1744684800000
cpu.user_seconds_total 234561.8 1744684800000
cpu.utilization_percent 12.4 1744684800000
files.max 9223372036854775807 1744684800000
files.open 1536 1744684800000
files.used_percent 0.0000000167 1744684800000
memory.available_bytes 8589934592 1744684800000
memory.total_bytes 17179869184 1744684800000
memory.used_bytes 8589934592 1744684800000
memory.used_percent 50.0 1744684800000
```

This format is deliberately close to the Prometheus exposition format — the de facto standard for metrics exchange. In Chapter 2, we will add full Prometheus compatibility, including histograms, summaries, and the `HELP` / `TYPE` metadata lines. For now, a human can `curl localhost:9100/metrics` and see the state of the machine.

1.10 Running Lens in Production

The agent compiles to a single static binary:

```
$ go build -ldflags="-s -w" -o lens .
$ ls -lh lens
-rwxr-xr-x 1 arjun arjun 5.2M lens
$ ./lens &
Lens v0.1 listening on :9100
$ curl -s localhost:9100/metrics | head -5
cpu.iowait_seconds_total 0 1744684800000
cpu.system_seconds_total 823 1744684800000
cpu.user_seconds_total 4521 1744684800000
cpu.utilization_percent 3.2 1744684800000
files.max 9223372036854775807 1744684800000
```

5.2 megabytes. Zero dependencies. Runs on any Linux kernel from the last decade. This is the power of Go for infrastructure software — a property we will rely on when Lens ships to 100,000 servers in Chapter 14.

1.11 What We Built

Lens v0.1 is a 300-line Go program that reads six kernel interfaces, produces 20+ metrics, and serves them over HTTP. It is not impressive. It is not novel. It is the simplest possible thing that answers the question "what is happening on this machine right now?"

And it is the correct foundation. Every observability platform — Prometheus, Datadog, New Relic, Grafana Agent — starts here: a binary on a machine, reading kernel counters, exposing them over HTTP. The difference between Lens and a

production observability platform is not architecture. It is the depth of the collectors, the efficiency of the storage, the sophistication of the query layer, and the scale of the deployment. All of that arrives in the chapters ahead.

1.12 The Principles So Far

1. Start with the kernel

Every useful metric originates in the Linux kernel. `/proc` is the interface. Understanding what the kernel exposes — and what it does not — is the difference between a metric that surfaces a problem and a metric that confirms a problem after a user reports it.

2. Cumulative counters, not instantaneous values

CPU time, disk bytes, network packets — the kernel reports these as counters since boot. Computing rates and percentages requires storing previous values and computing deltas. This pattern is universal.

3. The agent must be invisible

If your observability agent costs more CPU than the application it monitors, operators will disable it. Lens targets < 1% CPU and < 50 MB memory. This is not a nice-to-have. It is the defining constraint of observability infrastructure.

4. Text format, not binary

The `/metrics` endpoint returns plain text. A human can read it. A shell script can parse it. A monitoring system can scrape it. Text is the universal API — slower than binary, bulkier than Protobuf, but always debuggable with `curl` and `grep`.

5. The file descriptor leak was not a code problem

It was an observability problem. The service was healthy by every metric that OrbitPay was collecting. The gap was not in the code — it was in the metrics. Every observability platform is defined by what it chooses not to watch.

1.13 What's Next

Chapter 2 takes Lens from a snapshot viewer to a time-series database. We will implement ring buffers, downsampling, and compression for efficient metric storage, then compare our implementation to Prometheus's TSDB — the time-series database that powers the majority of the world's monitoring infrastructure. By the end, Lens will not just tell you what is happening now. It will tell you what happened five minutes ago, five hours ago, and five days ago.

> *The kernel knows everything about your system. It just needs to be asked. Every observability platform is, at its core, a very expensive way of reading files in* `/proc`.

End of sample

You have read the opening chapter of The Signal Path.

The complete book continues from here · every chapter, every appendix, in PDF and EPUB, with full companion code on GitHub.

EUR 49.00

Get the full book

https://djangozen.com/ebooks/book/the-signal-path/

Technovize Publishing