35 YEARS OF PERFORMANCE MANAGEMENT

Implementing OpenTelemetry with z/VM

Jonathan Castro

Software Engineer
Jonathan@velocitysoftware.com

Agenda

- What is observability and why does it matter?
- What is OpenTelemetry and are people/organizations using it?
- Our implementation with z/VM
- Improving efficiency

What is Observability?

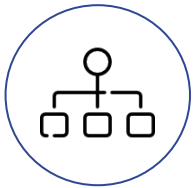
- **Observability is the ability to understand the internal state or condition of a complex system based solely on knowledge of its external outputs, specifically its telemetry.**

Metrics

Measurement of Service

Logs

Time Stamped Text Record



Traces

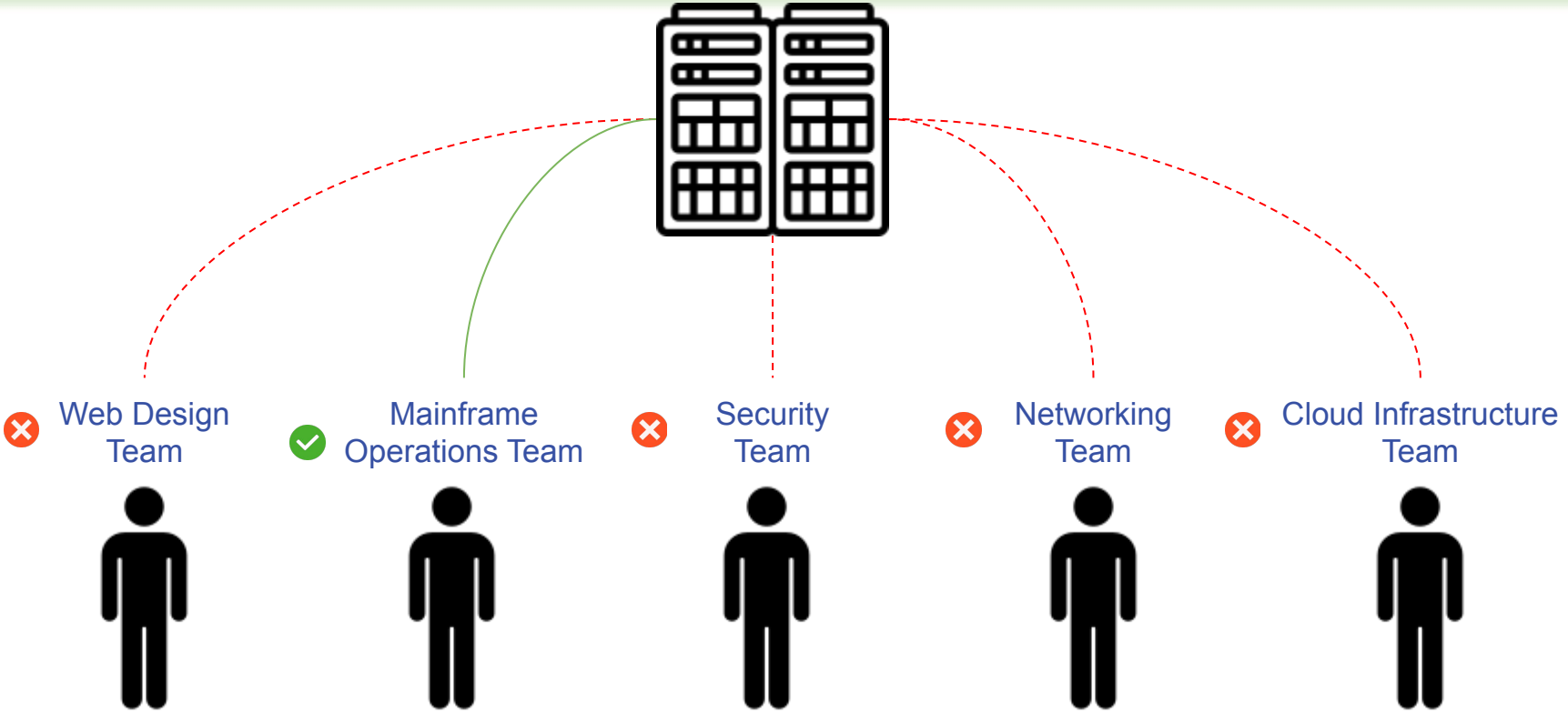
Full Path of Request

Source:
1 - https://www.ibm.com/think/topics/observability

Why Does Observability Matter?

- **Provides the data necessary to locate the root cause of system failures.**
- **Exposes outliers, allowing developers to optimize code before they impact users.**
- **Saves time by minimizing human intervention and only getting the required teams involved.**

Save Time

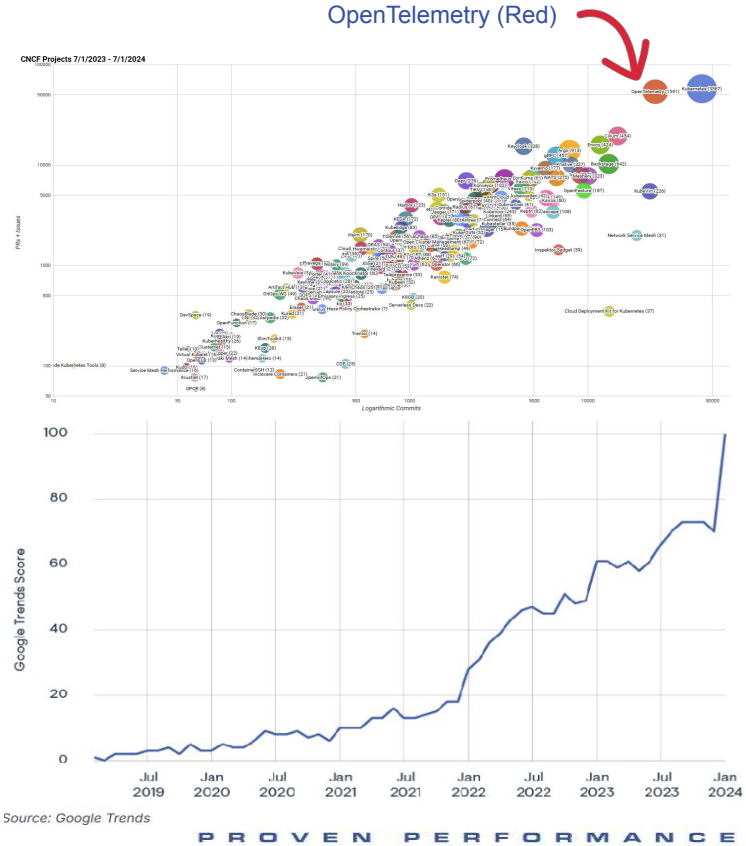

Web Design Team
Mainframe Operations Team
Security Team
Networking Team
Cloud Infrastructure Team

The Power of OpenTelemetry

- **Instrument Just Once:** The OpenTelemetry Protocol (OTLP) provides a standardized format that everyone uses, eliminating the need to rewrite code.
- **Vendor Agnostic:** We don’t need to know what vendor or observability backend we are using today, or may use in the future.
- **Focus on Data Quality:** Instead of fearing vendor lock-in, teams can safely spend time deeply customizing their data to make it rich and useful.

Gaining Popularity

- **Second Fastest growing project over the past 3 years according to CNCF.**
- **40% Increase in GitHub pull requests year to year, with contributions from thousands of individuals across 1,200 companies.**
- **100% increase in web searches year to year.**
- **A 410% increase in developer discussions on Stack Overflow.**



Sources:
1 - https://grafana.com/opentelemetry-report/
2 - https://www.cncf.io/blog/2024/07/11/as-we-reach-mid-year-2024-a-look-at-cncf-linux-foundation-and-top-30-open-source-project-velocity/

Adoption

Data from Grafana Labs Survey

(1300+ respondents across 76 countries)

65%

Are investing in both Prometheus & OpenTelemetry.

74%

Are using OpenTelemetry more or the same as last year.

90%

Are increasing or maintaining overall observability spending.

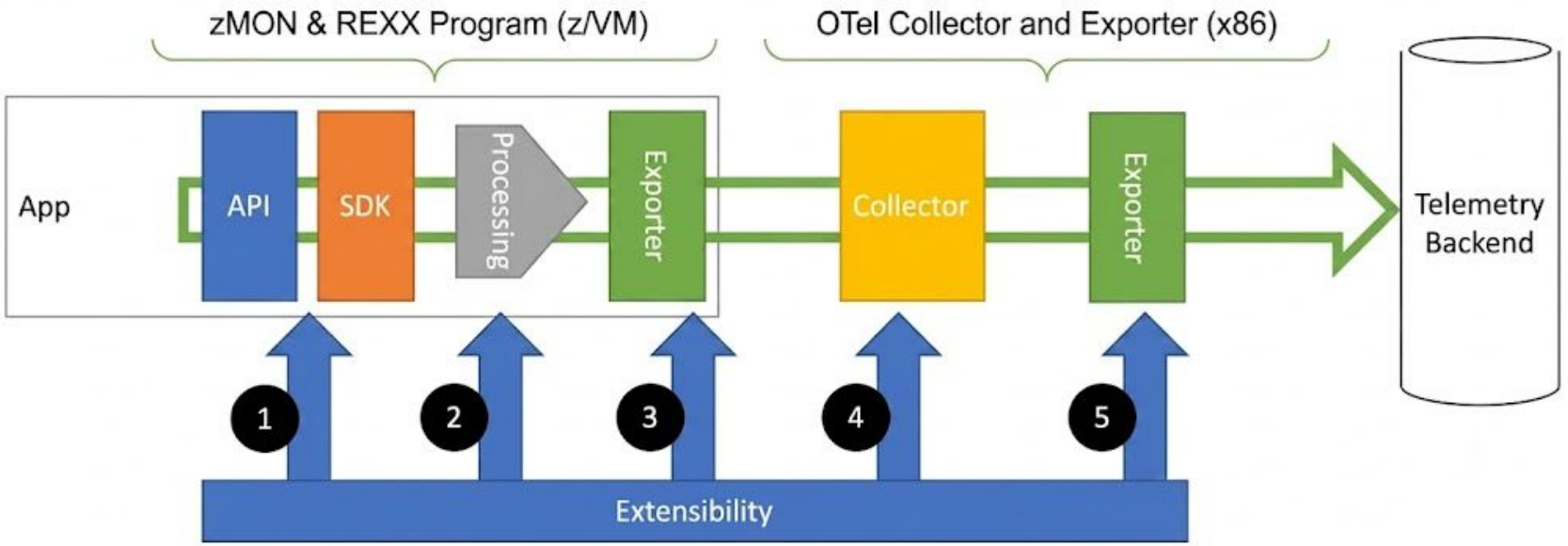
Top 2 User Problems OpenTelemetry Resolves

- Ease of Adoption and Management
- Ease of Switching Vendors and Backends

Source:
1 - https://grafana.com/observability-survey/?src=yt&mdm=social&cnt=description

Standard Flow of Data vs Our Implementation

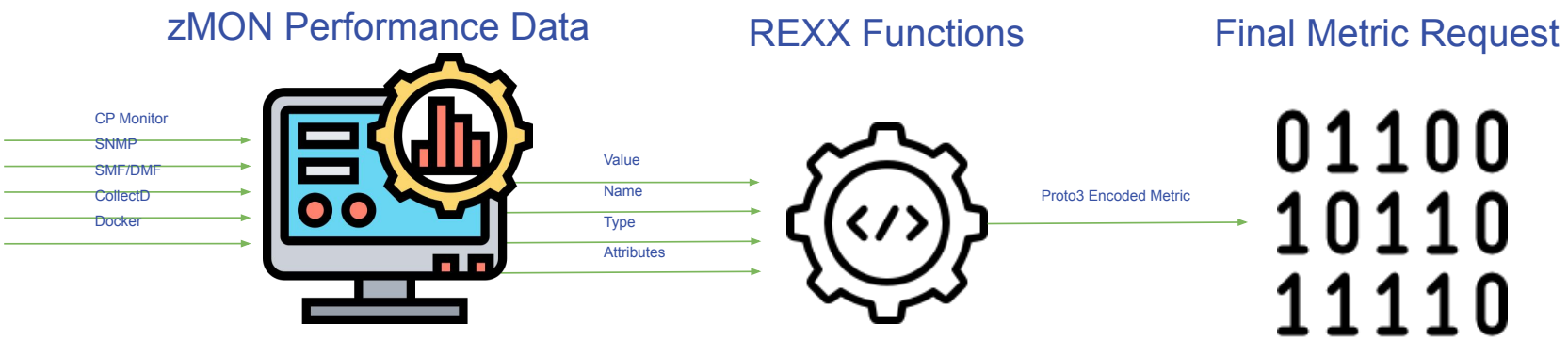
Standard Flow of Data



Source:
1 - https://lumigo.io/opentelemetry/opentelemetry-architecture/

Implementation

- **We parse our metrics through zMON, set important metric information, and finally encode our entire payload in proto3 binary format.**

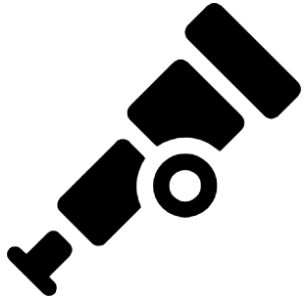


Implementation

- **Next is transmission, we send our payload to our collector by doing an HTTP POST request.**

Final Metric Request

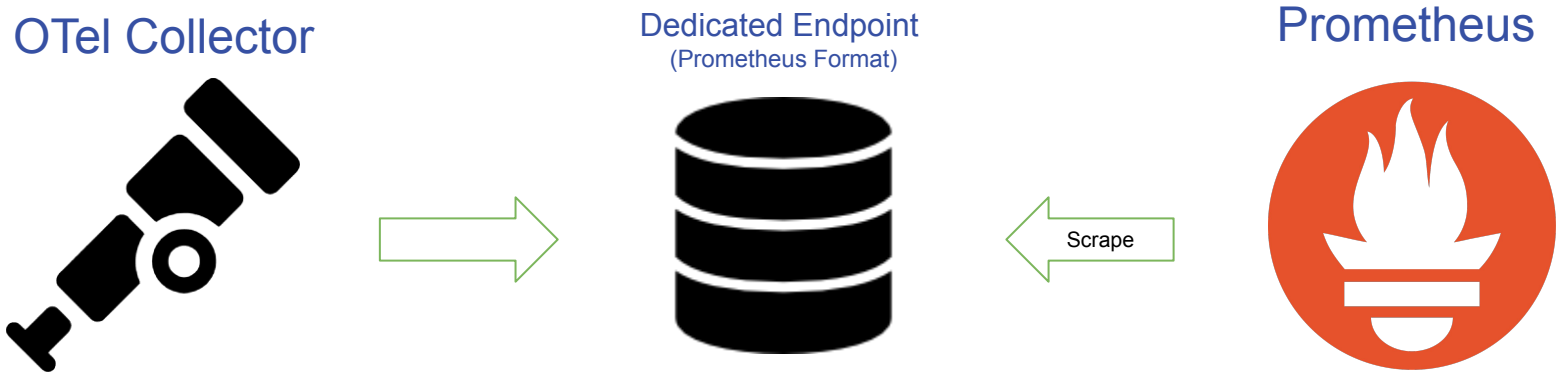
HTTP Post Request



OTel Collector

Implementation

- **Our collector now has our data, we use one or more of the 50+ exporters to route our data to almost any observability tool on the market.**



Implementation

- **Raw numbers are not enough on their own, each metric is tied with a System ID and a LPAR Name so we know exactly where our data is from.**

```
# HELP zvm_paging_rate
# TYPE zvm_paging_rate gauge
zvm_paging_rate{lparname="VSIVC1",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVC1"} 177.9620000000741
zvm_paging_rate{lparname="VSIVM2",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM2"} 0
zvm_paging_rate{lparname="VSIVM4",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM4"} 501.9669999974667
# HELP zvm_reconf
# TYPE zvm_reconf gauge
zvm_reconf{lparname="VSIVC1",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVC1"} 2048.1959999856917
zvm_reconf{lparname="VSIVM2",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM2"} 0
zvm_reconf{lparname="VSIVM4",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM4"} 0
# HELP zvm_runtime
# TYPE zvm_runtime gauge
zvm_runtime{lparname="VSIVC1",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVC1"} 60.00146000030395
zvm_runtime{lparname="VSIVM2",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM2"} 59.99998000026369
zvm_runtime{lparname="VSIVM4",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM4"} 59.984039999994025
# HELP zvm_spool_utilization
# TYPE zvm_spool_utilization gauge
zvm_spool_utilization{lparname="VSIVC1",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVC1"} 15.105999999920812
zvm_spool_utilization{lparname="VSIVM2",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM2"} 8.681000000037244
zvm_spool_utilization{lparname="VSIVM4",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM4"} 16.440000000093107
# HELP zvm_standby
# TYPE zvm_standby gauge
zvm_standby{lparname="VSIVC1",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVC1"} 1
zvm_standby{lparname="VSIVM2",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM2"} 1
zvm_standby{lparname="VSIVM4",otel_scope_name="",otel_scope_schema_url="",otel_scope_version="",systemid="VSIVM4"} 1
```

Implementation

- **We can bring this data to life using Grafana by building dynamic and interactive dashboards, which makes it much easier to spot bottlenecks in our data.**

Prometheus

Grafana Graphs

z/VM Overview

- 1 Attribute
- Only showing metrics for selected system

VSEn Overview

- 2 Attribute
- Uses chained variables

Improving Efficiency of Program

Improving Efficiency

- **Extracting telemetry data every minute requires a lot of processing power, we ran a profiler and dropped our most CPU intensive functions down to the system level.**

Before

```
133562  0.097746 EncodeVarint: procedure
 66781  0.056703   parse arg val
 66781  0.054586   out_hex = ""
 66781  0.046936
 97330  0.126182   do while val > 127
 30549  0.021514
 30549  0.041169     chunk = (val // 128) + 128
 30549  0.040735     byte = d2x(chunk)
 30549  0.037451     if length(byte) = 1 then byte = "0" || byte
 30549  0.030377     out_hex = out_hex || byte
 30549  0.031169     val = val % 128
 30549  0.021361
 30549  0.022818   end
 66781  0.047604
 66781  0.073212   last_byte = d2x(val)
166109  0.156840   if length(last_byte) = 1 then last_byte = "0" || last_byte
 66781  0.048513
 66781  0.169290   return out_hex || last_byte
```

After

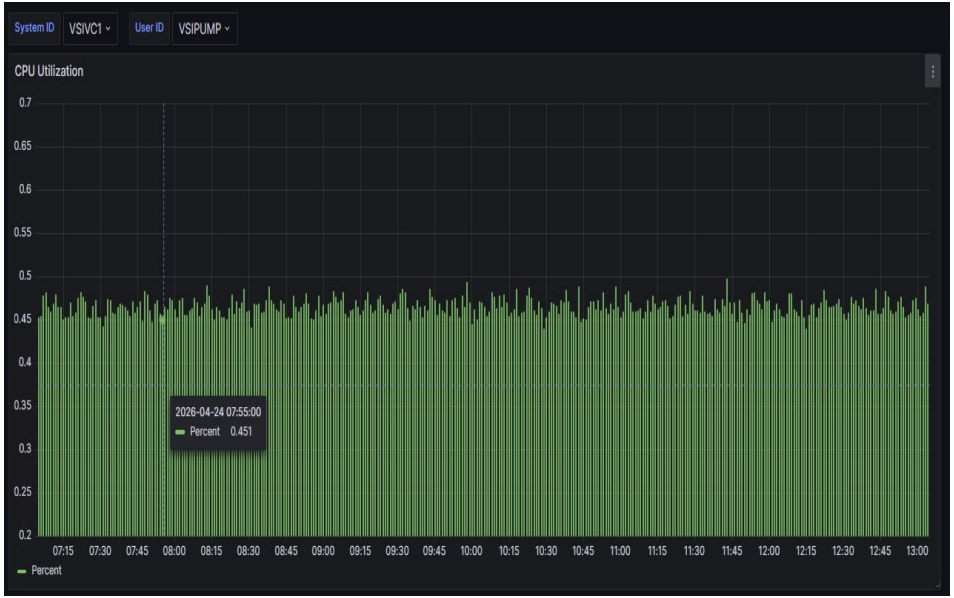
50% Decrease

```
1386   140890  0.107839 EncodeVarint: procedure
1387    70445  0.061437   parse arg val
1388    70445  0.052154
1389    70445  0.060409   varint_val = ""
1390    70445  0.214455   Address command 'ENVARINT' val
1392    70445  0.136434   return varint_val
```

Programs CPU Utilization

Before Assembler

~2300 Metrics / Minute



After Assembler

45% Decrease

Conclusion

Thank you for listening!

Jonathan Castro
Software Engineer
Jonathan@velocitysoftware.com