Observability Day

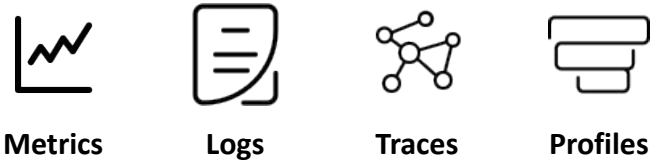
NORTH AMERICA

Observability Day
NORTH AMERICA

Perfect Match: Correlating Continuous Profiling with Distributed Tracing for Stronger Observability

Jonas Kunz & Christos Kalkanis

What is Profiling



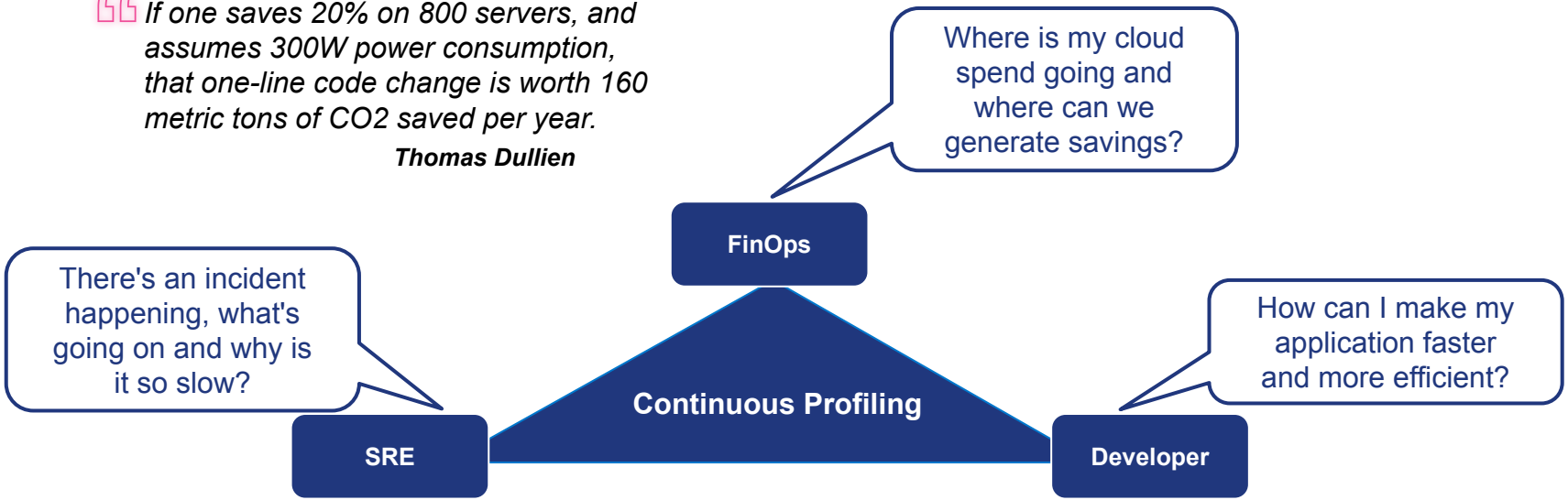
- The **fourth** pillar of Observability
- Different sources of profiling data: On-CPU, Off-CPU, Memory
- Instrumentation vs **Sampling**
- Sampling CPU profilers interrupt CPU at regular intervals and record execution state
- An **On-CPU** profiler records stacktraces for every CPU core that gets interrupted
- **StackTrace**: Snapshot of function calls at a point in time

- StackTraces can span **kernel**, **native** application and **higher-level** runtime code
- **eBPF** can enable low-overhead continuous profiling in production without application instrumentation
- **Continuous** profiling offers deeper level of visibility, can expose **unknown-unknowns**

Benefits of Continuous Profiling

- **Optimize Compute Efficiency**
- **Quickly Identify Performance Bottlenecks**
- **Save on Cloud Cost**
- **Reduce CO2 Footprint**

> *If one saves 20% on 800 servers, and assumes 300W power consumption, that one-line code change is worth 160 metric tons of CO2 saved per year.*
>
> ***Thomas Dullien***



Low-Overhead Continuous Profiling

Observability Day NORTH AMERICA

- Optimyze.cloud launches low-overhead multi-runtime zero-instrumentation profiler in 2021
- Acquired by Elastic soon after
- Donated to OpenTelemetry in 2024
- Continued development and evolution

opentelemetry-ebpf-profiler

ARM64 x86-64

Unobtrusive. Frictionless Deployment

Powered by eBPF. Requires **zero-instrumentation,** no code changes or app restarts. **Gain faster ROI.**

Whole-System Visibility

Unlock unknown-unknowns - from the kernel through userspace into high-level code, across **multi-cloud workloads.**

Polyglot Visibility

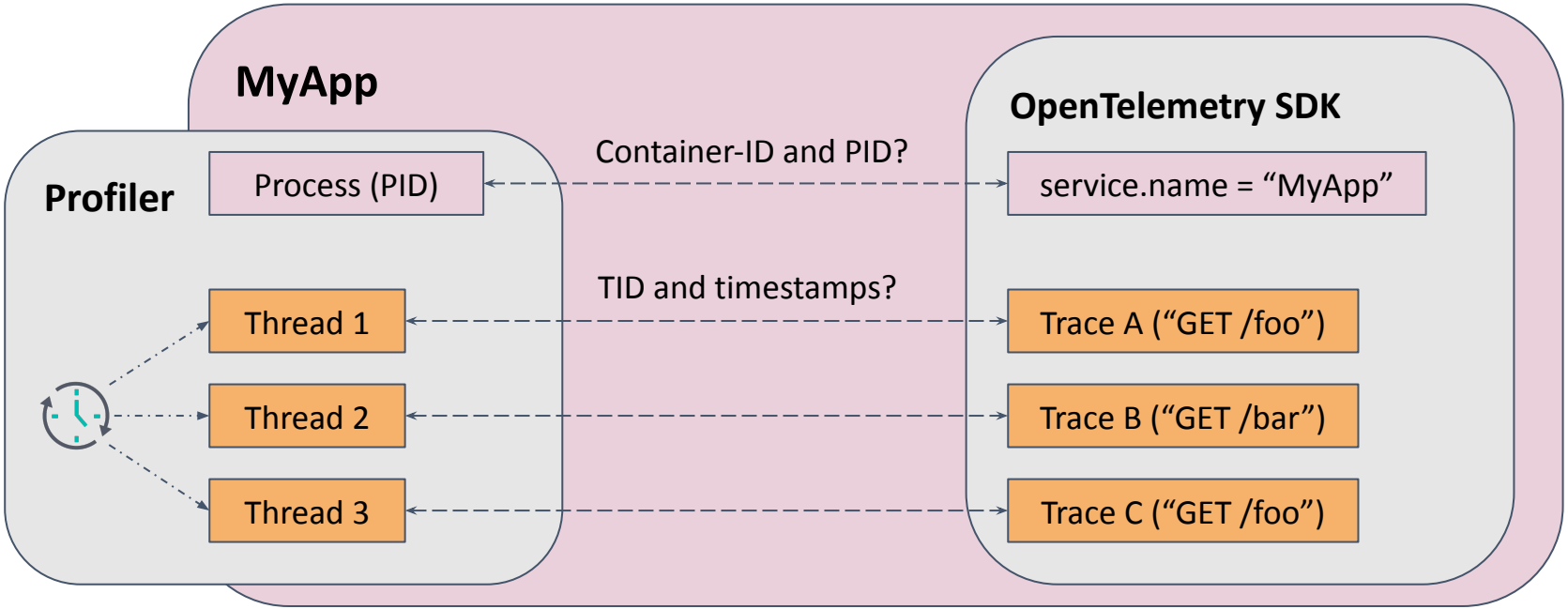
C/C++, Rust & Go (without debug symbols on host) PHP, Python, Java (or any JVM language), Ruby, DotNet, Perl & NodeJS.

Extremely Low Overhead

Continuous profiling in **production with negligible overhead**.

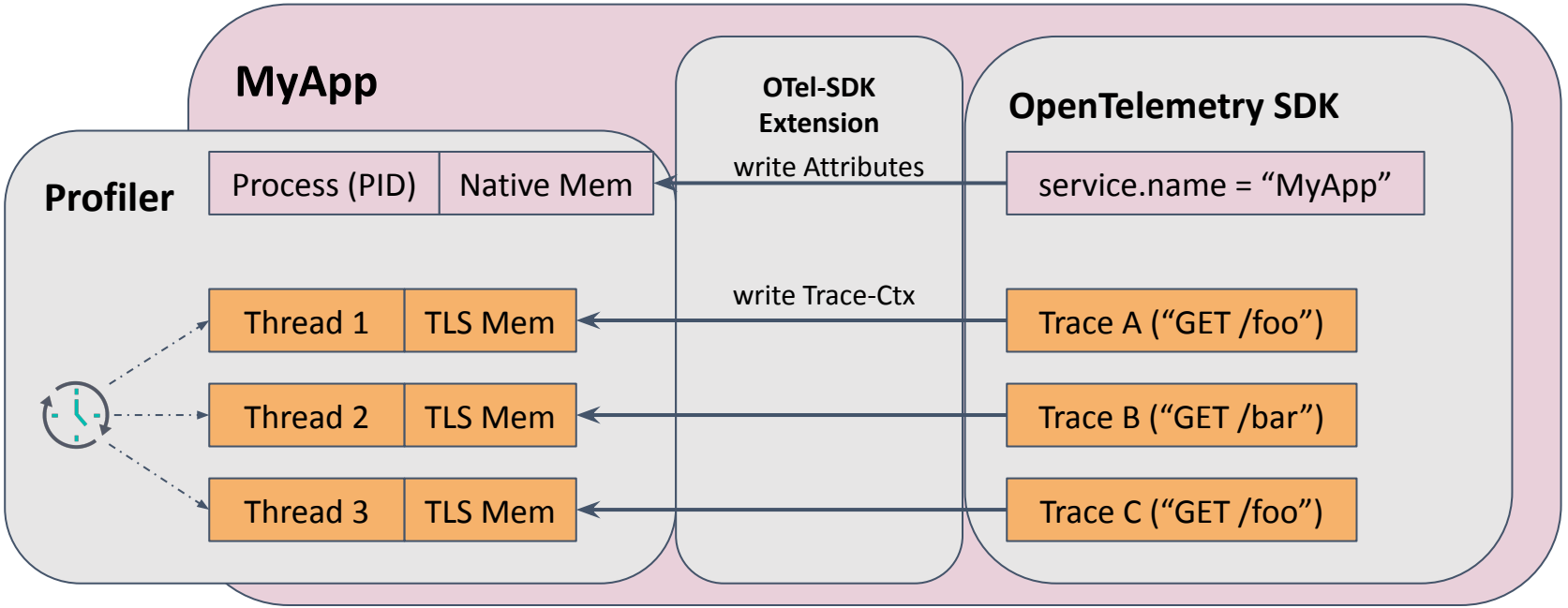
Typical case: < 1% system CPU, ~250MB of RAM

The Correlation Gap



- Root cause for CPU Usage of “MyApp”?
- Where does “GET /foo” spend its CPU-time?

The Correlation ~~Gap~~ Approach



- Root cause for CPU Usage of “MyApp”?
- Where does “GET /foo” spend its CPU-time?

Otel-Extension Implementation

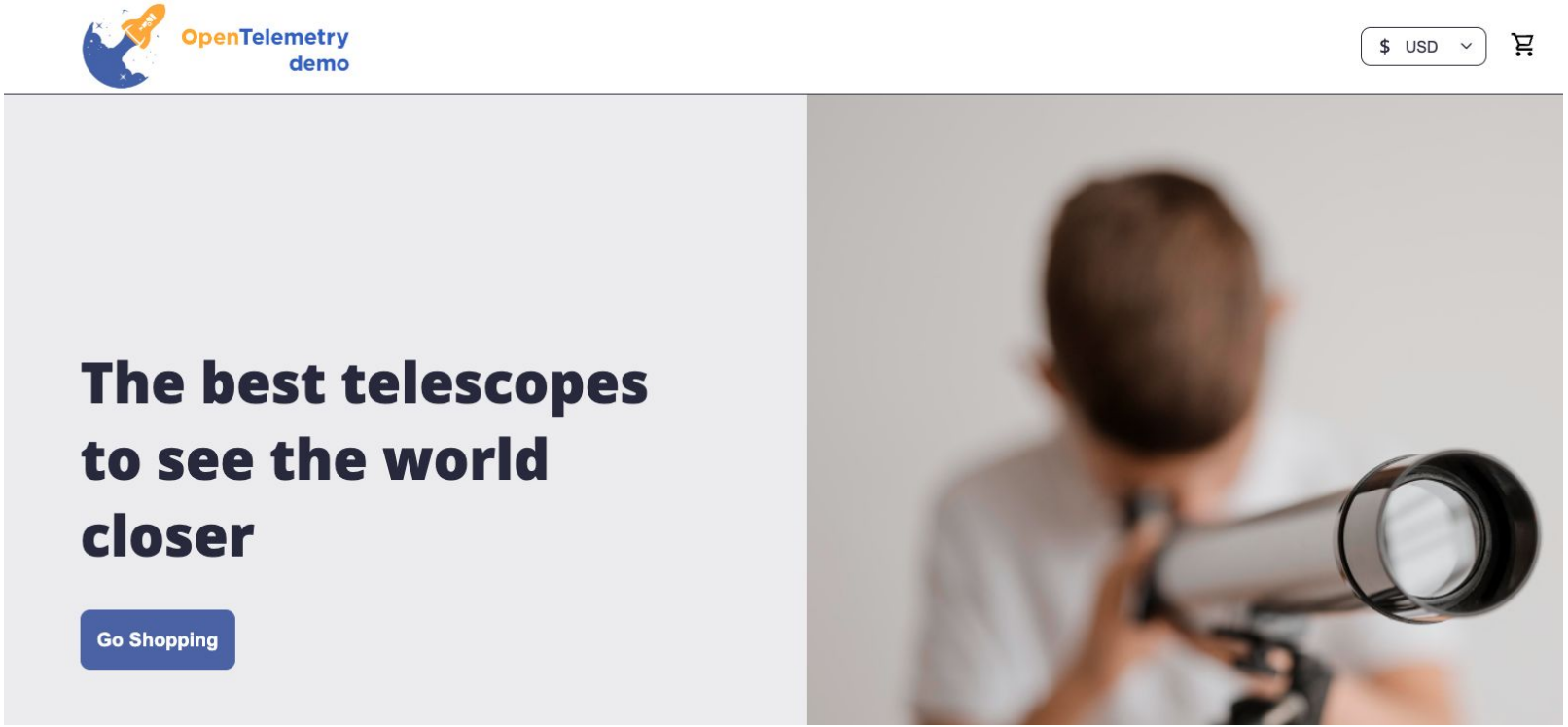
Java SpanProcessor available on GitHub:

```
Resource resource = Resource.builder()
   .put(ResourceAttributes.SERVICE_NAME, "my-service").build();

UniversalProfilingProcessor processor =
    UniversalProfilingProcessor.builder(exportingProcessor, resource).build();

SdkTracerProvider tracerProvider = SdkTracerProvider.builder()
  .addSpanProcessor(processor).build();
```

Loads a native JNI library which

- Populates a native, global var with resource-attributes
- Allocates native TLS
- Keeps the native TLS in sync on OTel context changes

Demo Time



https://opentelemetry.io/ecosystem/demo/

AdService with “Enhancements”

- AdService: Java App for serving advertisements
  - Random Ads
  - Targeted Ads: (E.g. for “telescopes”, “binoculars”, …)
  - Type recorded as span-attributes

- Problems:
  - Built-in: Background CPU-usage
  - Added:
    - Background allocations for GC-pressure
    - Code-path dependent CPU bottlenecks

Observability Day
NORTH AMERICA

Questions?