



KubeCon



CloudNativeCon

Europe 2026

Hacking GPU Observability: eBPF & Ephemeral Containers in Action on Kubernetes

Brandon Kang

Principal Technical Solutions Architect
Akamai Technologies, Cloud Technology Group



The GPU Era is Breaking Cloud-Native Toolkit



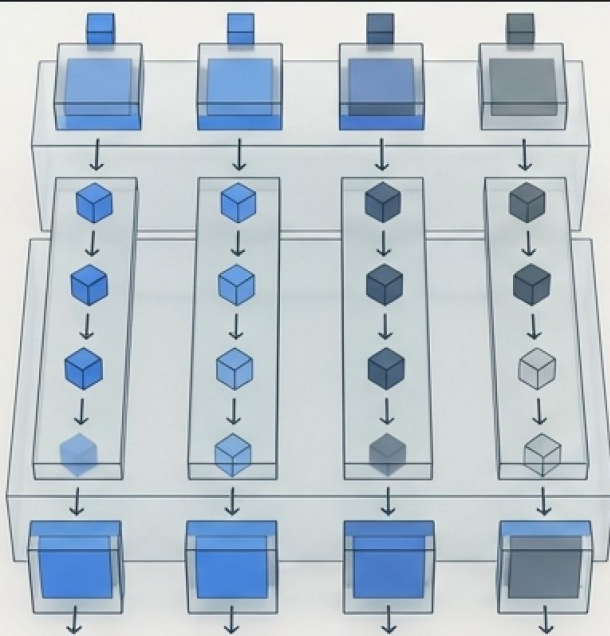
KubeCon



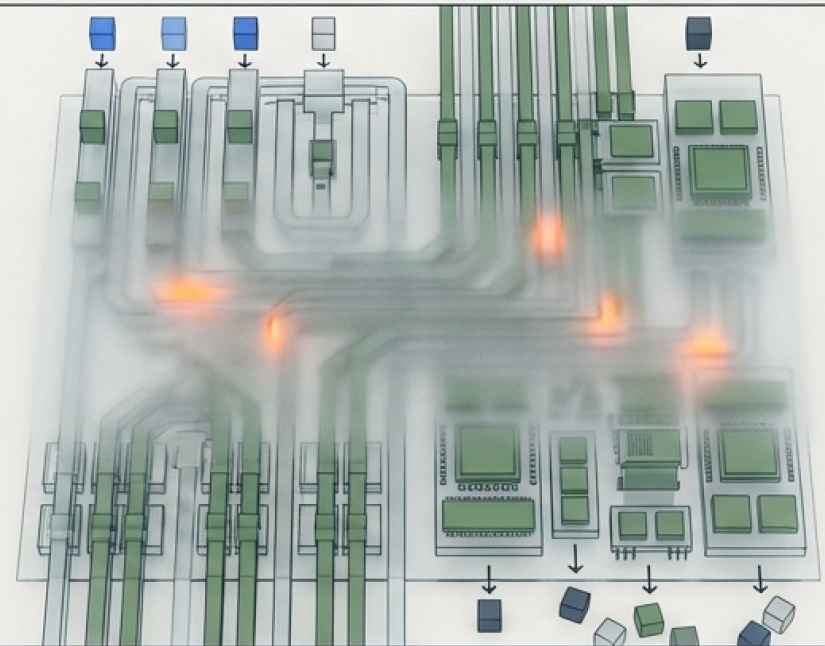
CloudNativeCon

Europe 2026

CPU Workload: Predictable



GPU Workload: The Alien Landscape



Hardware Reality: Operating GPUs introduces unfamiliar failure modes

CUDA Out-of-Memory (OOM) crashes

Frequent, silent failures at peak loads

Silent kernel stalls & thermal throttling

Unpredictable performance degradation

Scale Failure



GPU failures were one of the primary causes of training halts

Visibility Illusion: H/W metrics vs. Application insight

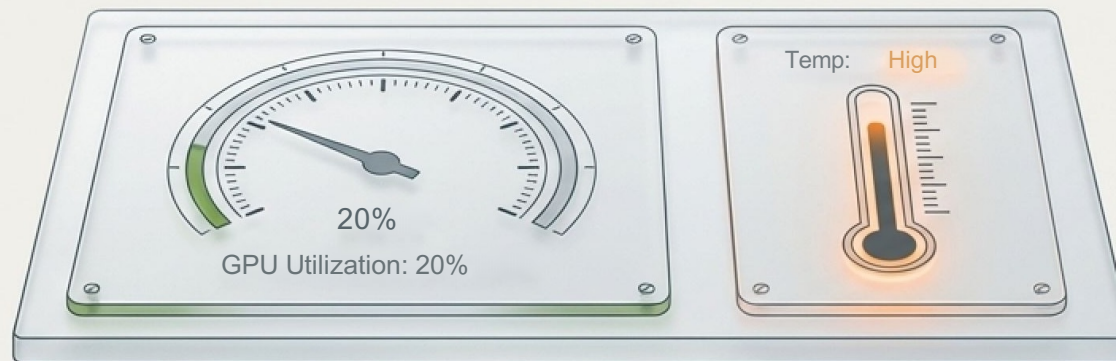


KubeCon

CloudNativeCon

Europe 2026

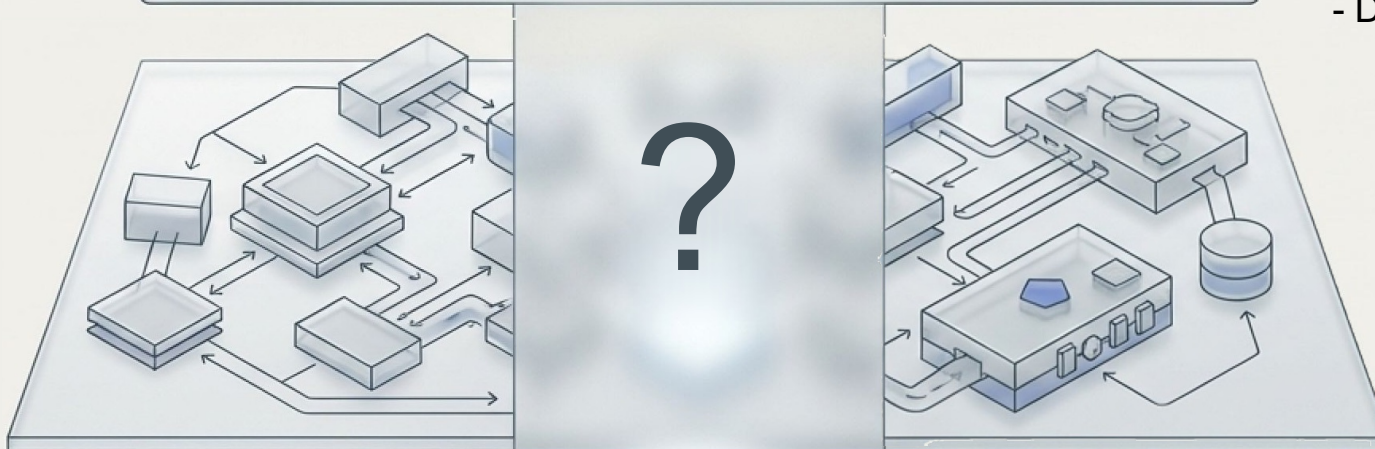
The Status Quo



Tools like NVIDIA DCGM & nvidia-smi show **WHAT** happened at a high level but not **WHY**

e.g.)

- Kernel launch overhead
- PCIe bottleneck
- Data pipeline issue

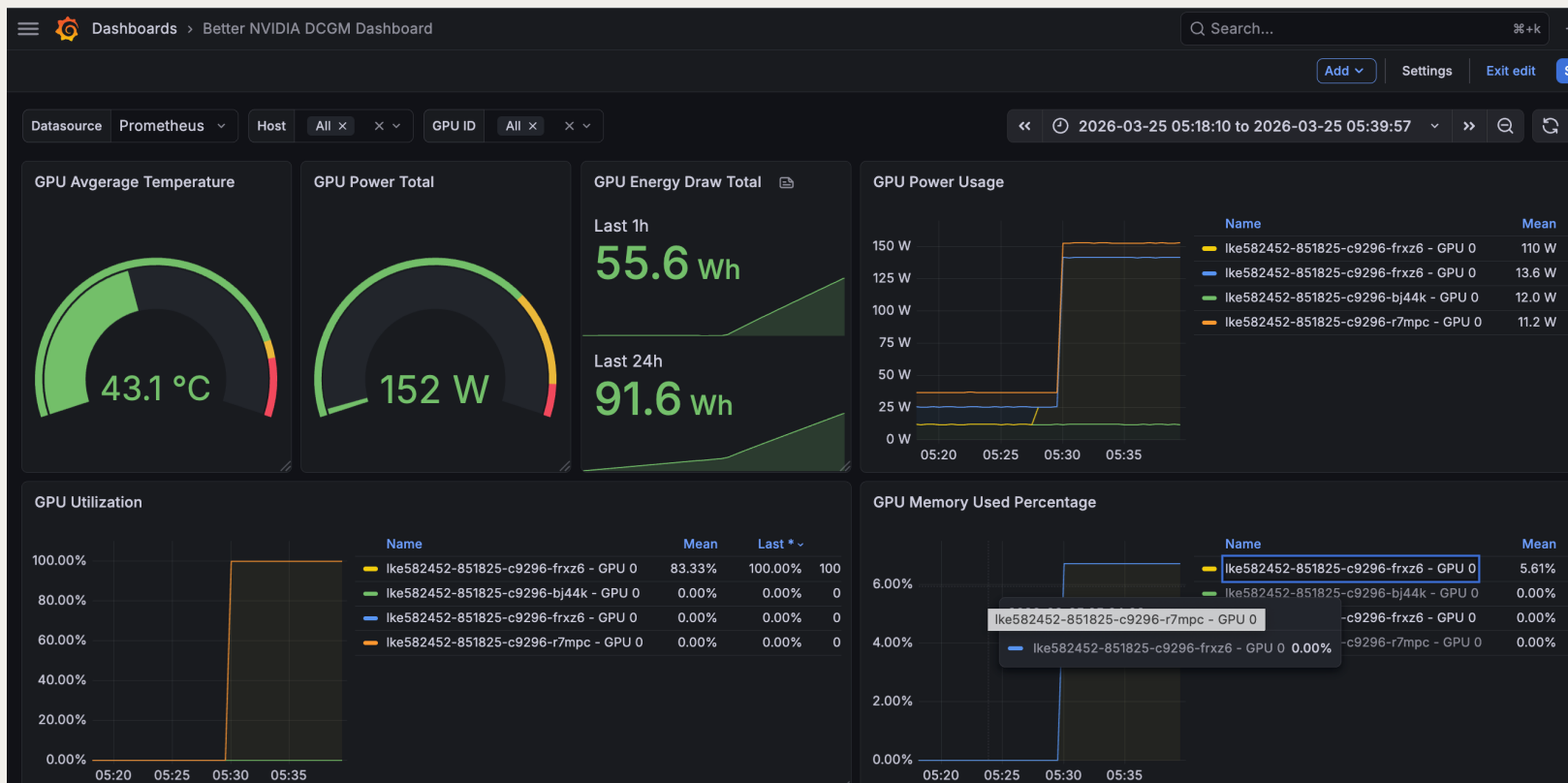


Same GPU, Same Model
→ Different Utilization

Why?

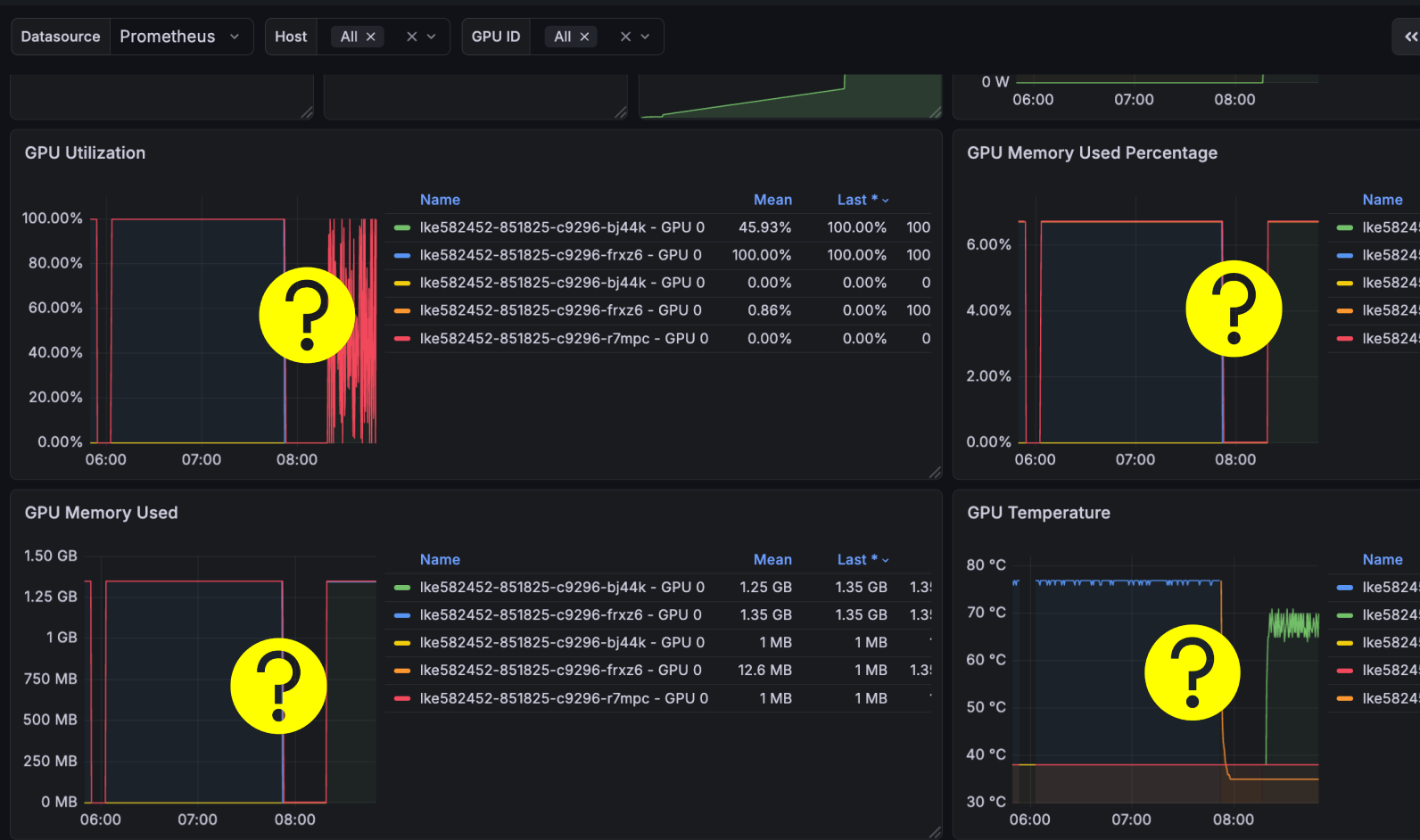
The blind spot: application logic, data ingestion, and bottlenecks are hidden

DCGM(Data Center GPU Manager)



- Compute / Utilization
- Memory
- Power / Energy
- Temperature
- Clocks / Performance
- Reliability / Errors

DCGM(Data Center GPU Manager)



Anatomy of an LLM inference workload



KubeCon



CloudNativeCon

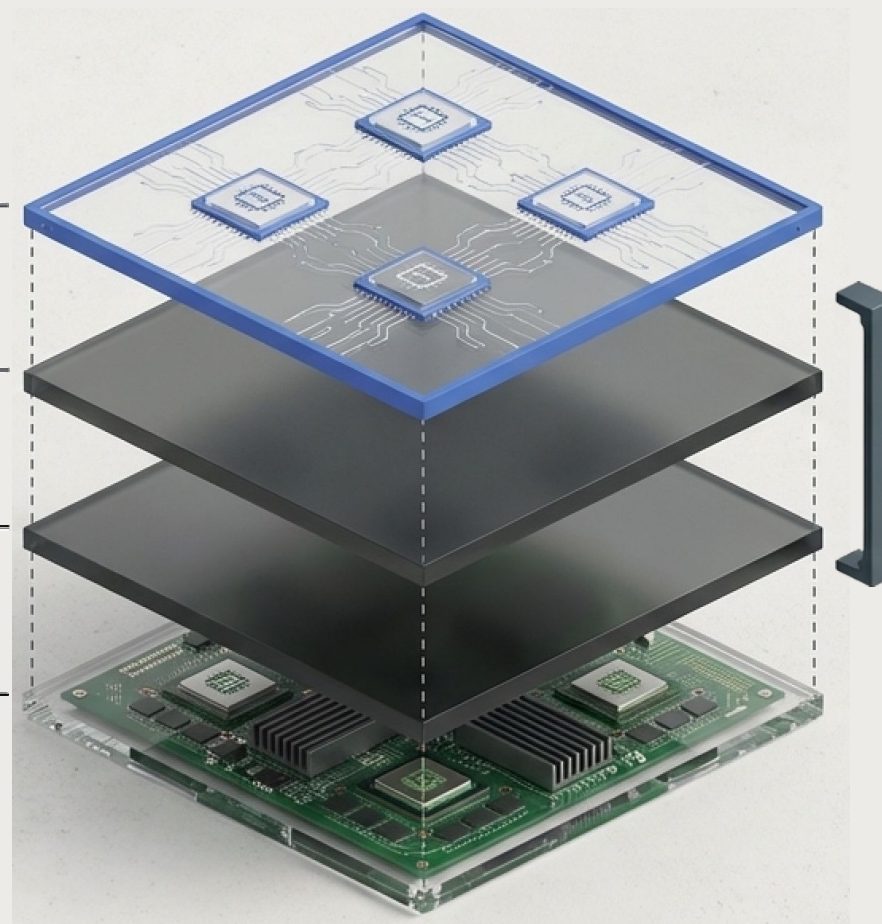
Europe 2026

Application
(Inference Server / PyTorch)
The CPU (via runtime/framework)
acts as the orchestrator

CUDA Runtime

CUDA Driver

GPU Hardware
The parallel execution engine

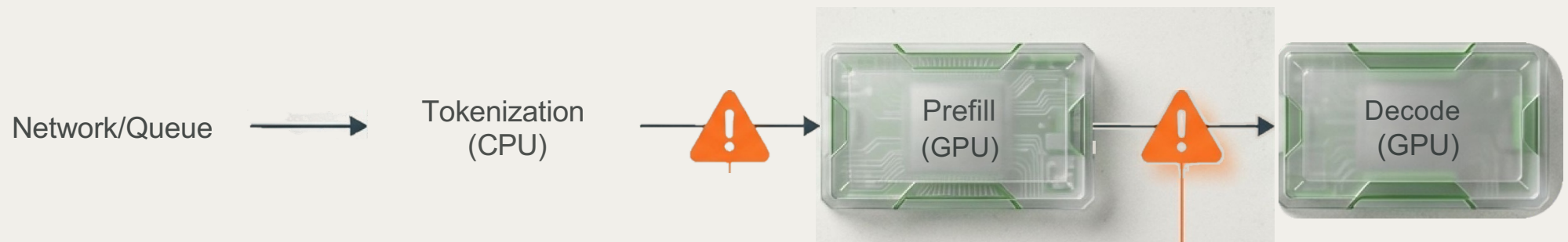


The Blind Spot:

Traditional tools monitor the top and bottom layers

Bottlenecks and security anomalies live exclusively in the shadowed boundary between the Driver and the Runtime

Performance bottlenecks hiding in the shadows



PCIe Transfer Bottlenecks

Frequent CPU <-> GPU memory copies cause high PCIe bandwidth usage but low GPU utilization. The GPU is simply waiting for data

Kernel Launch Overhead

Transformer models launch thousands of tiny math functions (CUDA kernels). Each launch typically incurs a 5–20 microsecond delay

An example) A model might exhibit 48ms of latency just from kernel fragmentation, keeping utilization artificially low (around 30%)

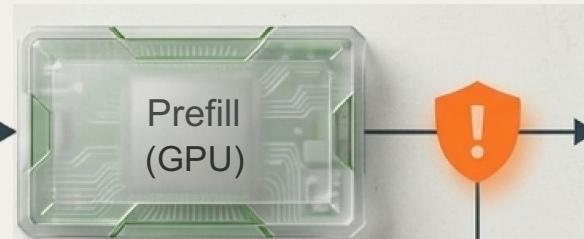
Security anomalies hiding in the same shadows

Rogue Kernel

Network/Queue



Tokenization
(CPU)



Decode
(GPU)

Unexplained cudaMemcpy Spikes

An anomalous spike in Device-to-Host memory copies isn't necessarily a poorly optimized application—it can be a prime indicator of potential model data exfiltration

Unauthorized Execution

Unmapped kernel launches indicate rogue workloads (like unauthorized crypto-mining) hijacking idle GPU cycles within a compromised pod

The Security Blind Spot:

If you cannot observe the CUDA Runtime, you cannot secure container provenance

Legacy profiling

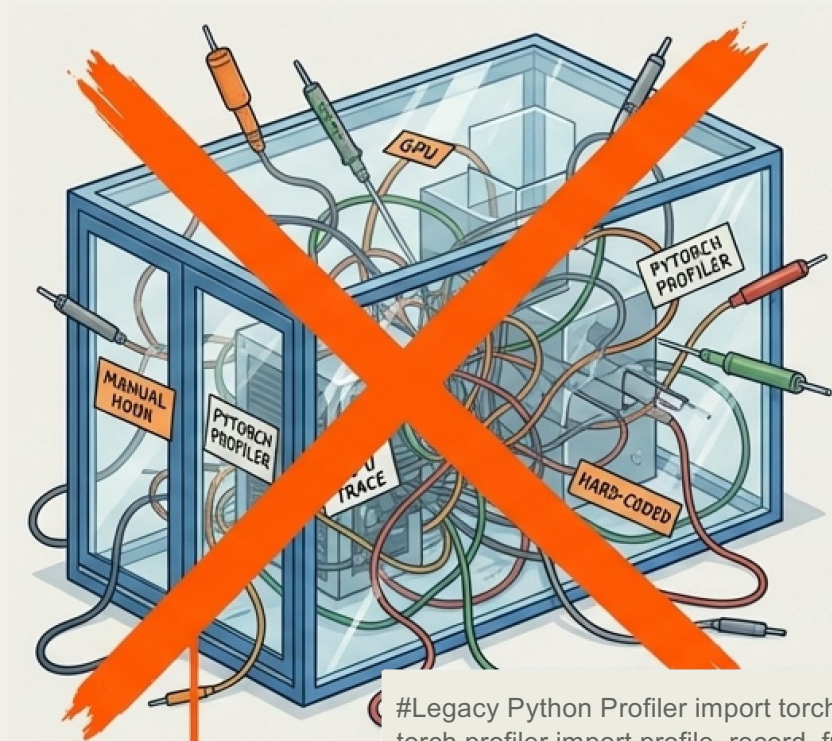


KubeCon



CloudNativeCon

Europe 2026



```
#Legacy Python Profiler import torch from  
torch.profiler import profile, record_function
```

```
with profile (...) as prof: with  
record_function("model_inference"):  
model(input_data)  
prof.export_chrome_trace("trace.json")
```

Heavy Instrumentation

Requires developers to manually annotate specific functions in Python code (e.g., PyTorch profiler, Nsight)

Breaks Immutability

Forcing debug tools into production apps violates immutable container patterns

Performance Penalty

Adds massive overhead to the application process skewing production metrics

Vendor Lock-In

Switching from NVIDIA to AMD GPUs requires rewriting the entire profiling framework

eBPF (Extended Berkeley Packet Filter)



CloudNativeCon

Europe 2026



“eBPF does to the Linux kernel
What JavaScript does to HTML”

[Brendan Gregg](#)

eBPF (Extended Berkeley Packet Filter)



KubeCon



CloudNativeCon

Europe 2026

bpftool/eBPF Tools

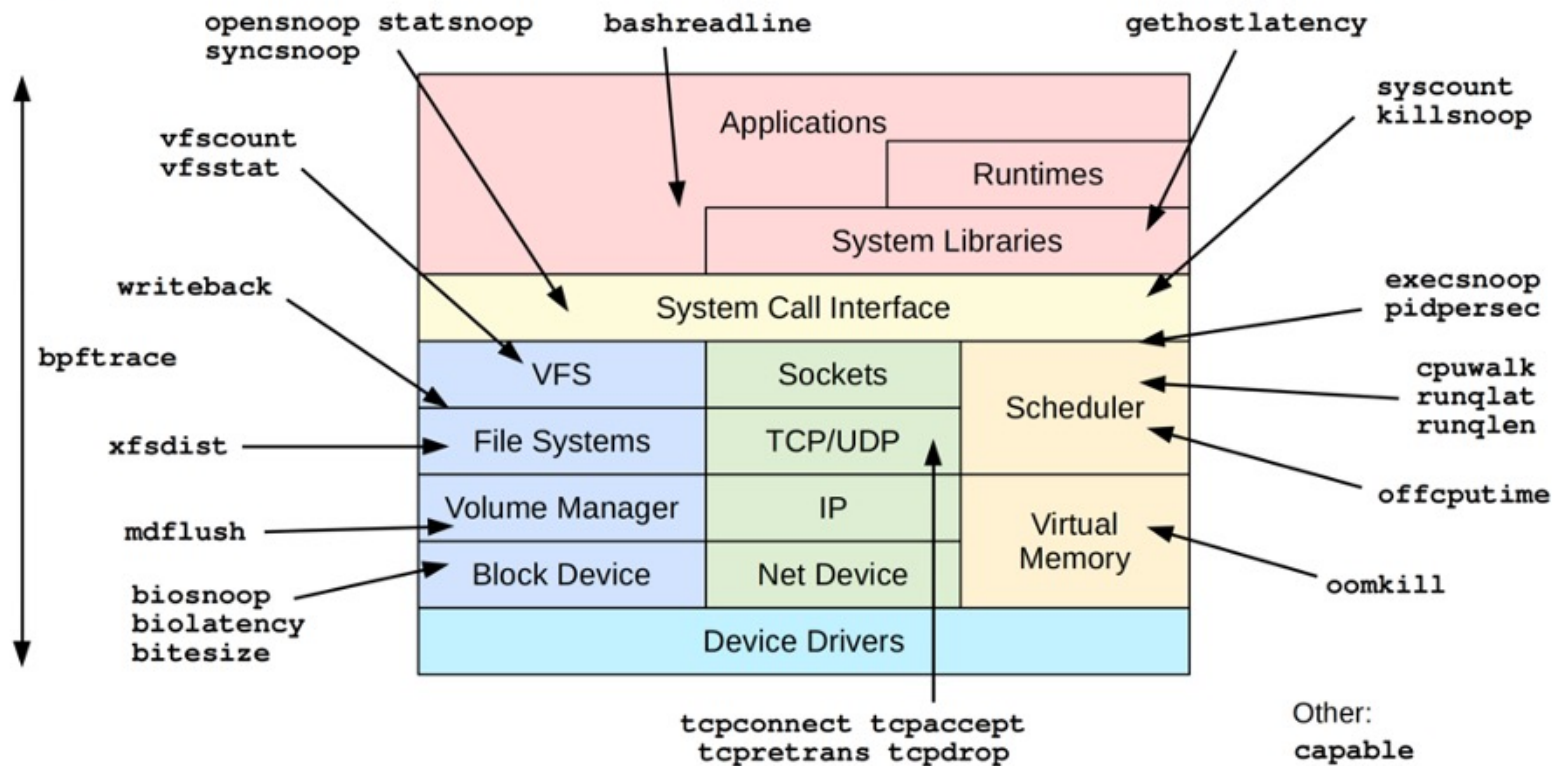


Diagram by Brendan Gregg, early 2019. <https://github.com/iovisor/bpftool>

Enter eBPF: Zero-Code kernel X-Ray vision



KubeCon



CloudNativeCon

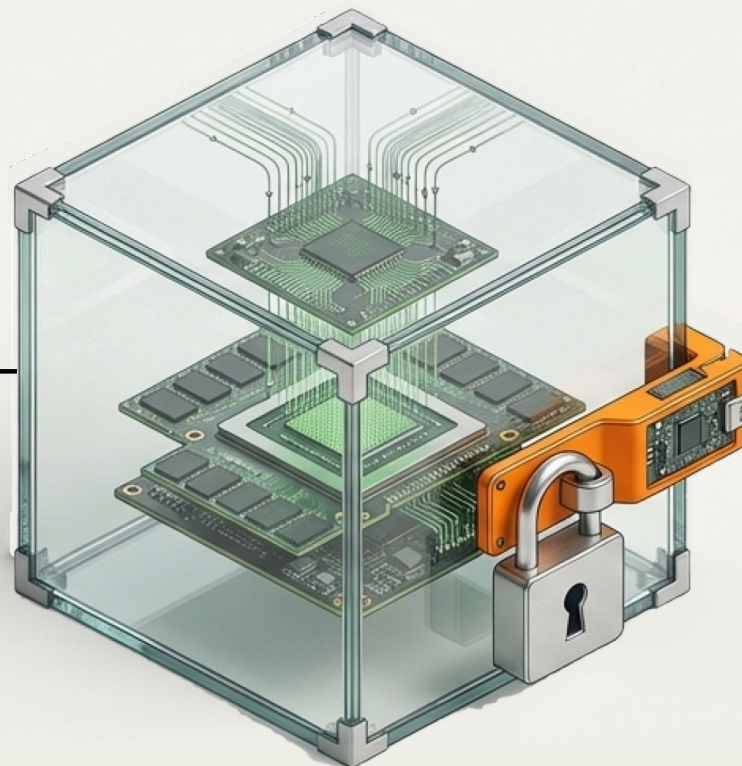
Europe 2026

The Mechanism

eBPF allows safe, low-overhead program execution directly within the Linux kernel sandbox

Zero Instrumentation

Intercepts underlying CUDA driver calls without requiring a single modification to the application code



Framework Agnostic

Whether the application is TensorFlow, PyTorch, or a custom binary,
All must pass through the Linux kernel to reach the GPU driver
eBPF catches them all

Tracing the CUDA runtime: eBPF in action



KubeCon



CloudNativeCon

Europe 2026

kthread	runtime.goexit										run	r	n	ne	git	k8s	g	n	runtime.goexit				ru	r	runtime.gcB				github.com/containerd/cc	0x9	runt	m		run	r	0:
kswa	wo	google.golang.	golang.org/x/sync	go.opentelemetry.io/ebpf	ri			r	c	ne	git	k8s	g	n	net/ht	google.go	net/h	ne		ri	runtime	run			github.com/containerd/cc	0	ne	ξ	ξ				ri	0:		
bala	pr	google.golang.	github.com/parca-	go.opentelemetry.io/ebpf	ri			r	c	ne	git	k8s	g	n	net/ht	google.go	net/h	ne		r	runtime	ru			github.com/containerd/cc	0	ne	ξ	ξ				r	0:		
shri	v	golang.org/x	github.com	github	go.opentelemetry.io/ebpf	ri			r	c	ne	git	k8s		n	net/ht	google.go	?	ne	ne	i			ru		github.com/containerd/c		ne	r	ξ				0:		
shri	v	google.golan	github.cc	github	go.opentelemetry.io/ebpf						i	ne	g:	k8s		n	net/ht	google.go	n	?	ne			ru		github.com/containerd/c		n								
shr	v	golang.org/x	github.cc	github	go.opentelemetry.io/ebpf						i	ne	g:	k8s		t	net/ht	github.co	ic	ne	ne			ri		path/filepath.Walk		n								
shr	-	golang.org/	github.cc	g bu	go.opentelemetry.io/ go							ne	ξ	k8s		k	net/ht	google.go	n	ic	ne					path/filepath.walk		n								
shr	v	google.gola	?	g os	go.opentelemetry.io/ gi							ne		k		k	bufio.	github.co	n	ne	ne					path/filepath.walk		n								
		google.go:	github.c	g os	go.opente	go.opente	gi					ne		k		k	net/ht	github.co	ir	n	ne					path/filepath.walk		n								
	c	net.(*conr	io.CopyB	g in	go.opente	go.opente						ne				g	net/ht	github.co	?	i						path/filepath.walk		n								
	c	net.(*netf	os.(*Fil	g ?	go.opente	github.c						ne				n	net.(*	github.co	ir	?						path/filepath.walk		n								
		internal/f	io.Copy	g in	gc go.op	github.c						ne				n	net.(*	github.co	sy	i						path/filepath.walk		n								

- Application (User S
- System

DCGM shows what happens.
eBPF helps explain why it happens.

- Application Layer
(User Space Behavior)
- System Call Layer
- Kernel Layer
(OS & Scheduler)
- I/O and Data Movement
- GPU Driver Interaction Layer
- Container / Kubernetes Layer

entry_SYSC	interne	g x6	internal	s)	g	interr	os.(*File	)	entry_ξ	ξ
do_syscall	syscall	--	entry_SY	ir	g	entry_?	-		do_sysc	ξ
x64_sys_cr	syscall	ks	do_sysca	er	g	do_sys	os.generi	h	x64_sy	ξ
__x64_sys	syscall	vf	x64_sys_	dc	n	x64_sy	io.CopyBu	\	__x64_	ξ
ksys_writ	interne	ne	__x64_sy	xt	g	__x64_	os.(* gi	r	__do_s	ξ
vfs_write	entry_ξ	ex	__sys_bp	--		ksys_w	os.(* gi	ξ	vfs_s	>
new_sync_v	do_sysc	ge	bpf_map_	--		vfs_wr	inter: g:	ξ	user	-
sock_writ	x64_sys	fi	generic_	--		new_sy	sysca g:	ξ	fi	-
__sock_ser	__x64_ξ	co	__	ir		sock_w	inter: g:	1	pa	-
inet_sendr	ksys_wr	--	__	--		__sock	sysca g:		l:	-
tcp_sendm	vfs_wri	t				inet6_	sysca g:			-
tcp_sendr	new_syr					tcp_se	sysca g:			1
sk_pi	tc	ext4_fi				tcp_	inter: g:			1
skb_i	ext4_bu					tcp_	entry g			

- I/O and
- GPU D
- Contai

Tracing the CUDA runtime: eBPF in action

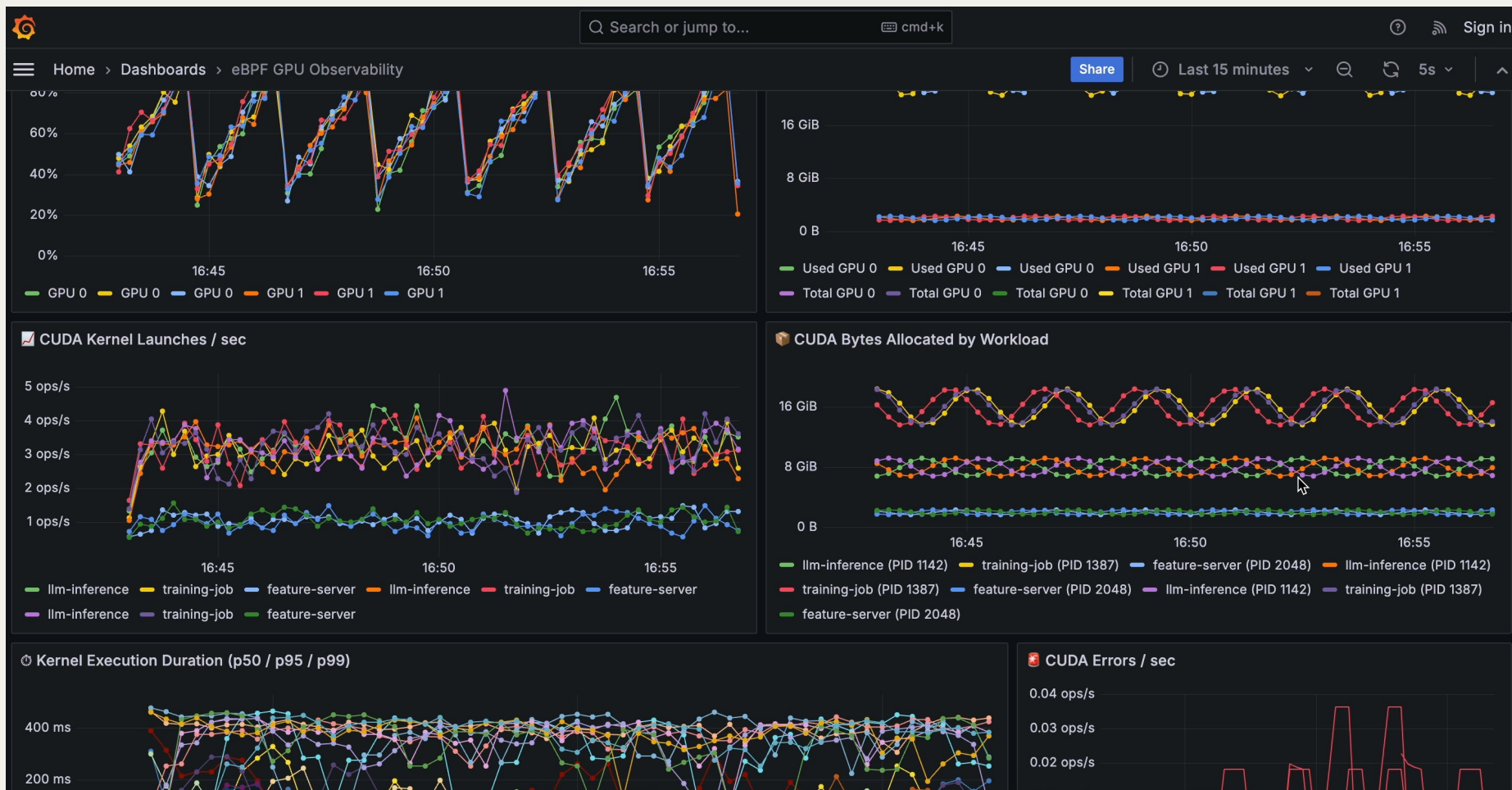


KubeCon



CloudNativeCon

Europe 2026



Tracing the CUDA runtime: eBPF in action

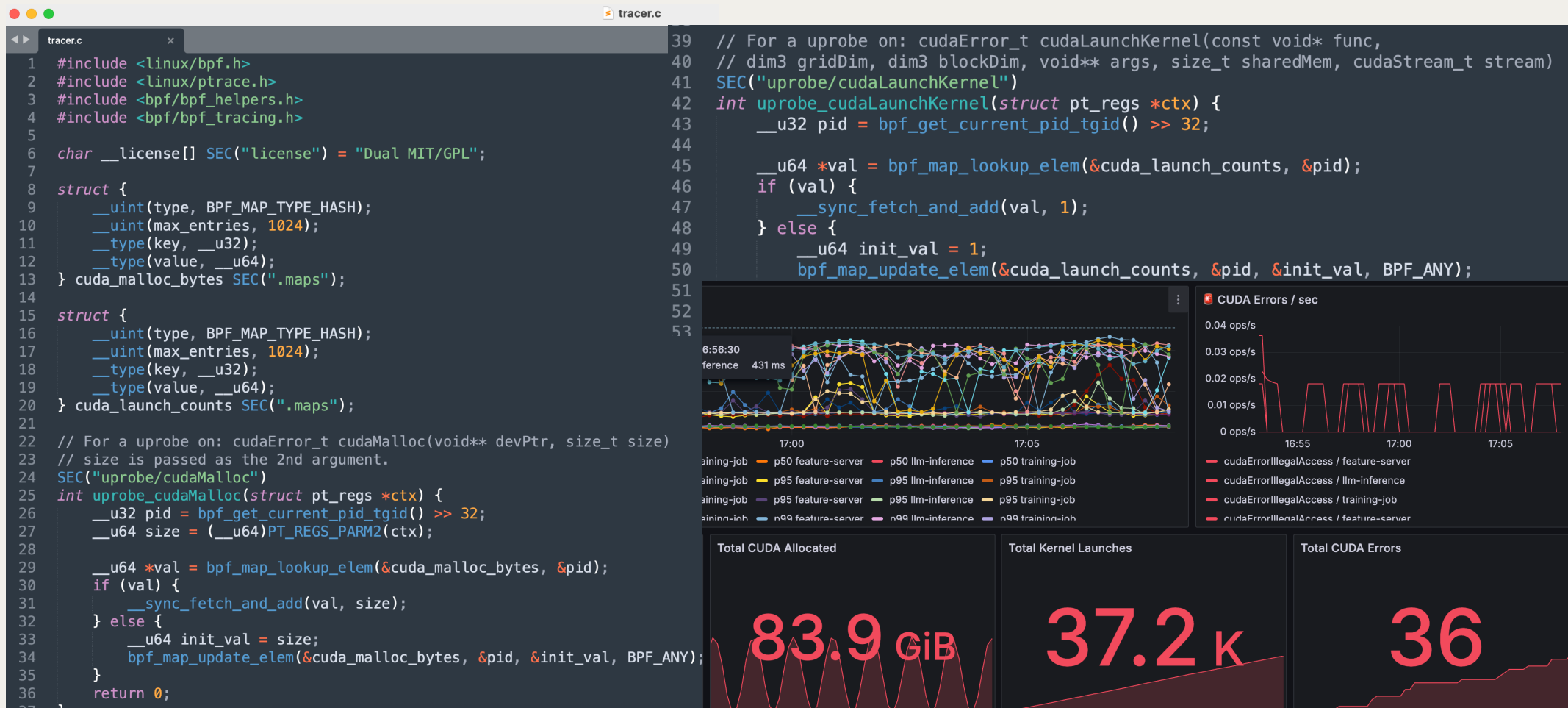


KubeCon



CloudNativeCon

Europe 2026



Tracing the CUDA runtime: eBPF in action



KubeCon



CloudNativeCon

Europe 2026

Here's what each file does:

`agent/bpf/tracer.c` — The eBPF Program (runs in the Linux kernel)

This is a **C program that runs inside the Linux kernel** via the eBPF virtual machine. You can't call it directly — it gets compiled into BPF bytecode and injected by the Go program.

What it does:

- Defines two **BPF maps** (key-value stores shared between kernel and userspace):
 - `cuda_malloc_bytes` — stores {PID → total bytes allocated}
 - `cuda_launch_counts` — stores {PID → number of kernel launches}
- Attaches two **uprobes** (userspace probes):
 - `uprobe/cudaMalloc` — fires every time any process on the node calls `cudaMalloc()`. It reads the `size` argument (2nd parameter via `PT_REGS_PARM2`) and adds it to the map for that PID.
 - `uprobe/cudaLaunchKernel` — fires every time any process calls `cudaLaunchKernel()` and increments the counter for that PID.

Think of it like a tap placed on a pipe — completely invisible to the application, no code change needed.

`agent/main.go` — The Go Loader (runs in userspace)

This is the **orchestrator** that bridges kernel space (the BPF program) and the outside world (Prometheus).

What it does, step by step:

- `rlimit.RemoveMemlock()` — Removes the memory lock limit so BPF maps can be created (required on most kernels)
- `loadBpfObjects()` — Loads the pre-compiled BPF bytecode (embedded at build time by `bpf2go`) into the kernel
- `link.OpenExecutable(libcudartPath)` — Opens `libcudart.so` on the host filesystem
- `exec.Uprobe("cudaMalloc", ...)` — Attaches the BPF program to the `cudaMalloc` symbol inside that library
- `http.Handle("/metrics", promhttp.Handler())` — Starts an HTTP server exposing Prometheus metrics on `:8000`
- Poll loop every 2s** — Iterates the BPF maps and updates Prometheus `Gauge` values so Prometheus can scrape them

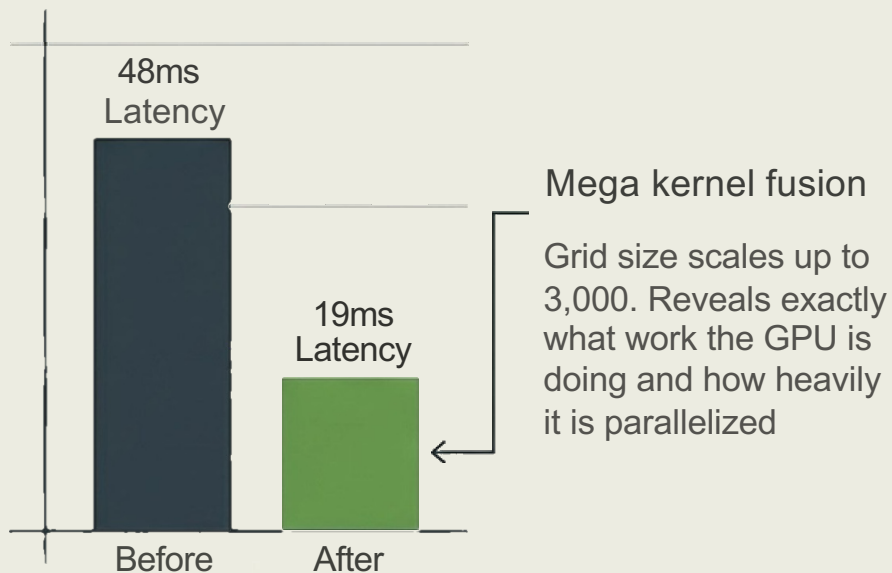
tracer.c (kernel)	main.go (userspace)
BPF maps	Poll maps → Prometheus Gauges → :8000/metrics
Uprobes fire	Uprobe attach via cilium/ebpf

The two files are inseparable — `tracer.c` collects the raw events at kernel speed, `main.go` makes that data queryable.

Tracing the CUDA runtime: eBPF in action

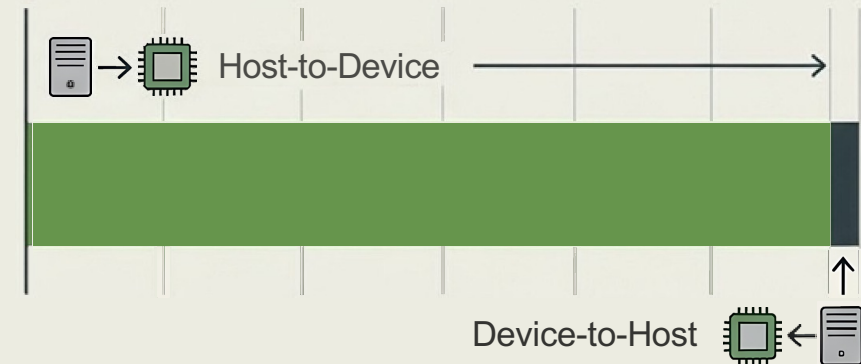
Execution Tracking: cudaLaunchKernel

Extracts: Grid Cardinality & Block Dimensions



Data Movement: cudaMemcpy

Extracts: Host-to-Device vs Device-to-Host metrics



Safety Orange

A sudden spike in Device-to-Host transfers flags an immediate security/exfiltration anomaly

Security anomalies hiding in the same shadows

	Traditional GPU Monitoring (DCGM/Profilers)	Cloud-Native Observability (eBPF)
Visibility Scope	Hardware-only (Temperature, Power, Global Util)	Deep runtime + kernel-level visibility
Instrumentation Burden	Heavy manual code changes required	Zero-code, automatic kernel attachment
Ecosystem Lock-in	Strictly vendor-locked (NVIDIA or AMD specific)	Vendor and framework agnostic
Security Context	Blind to process origin	Process-level provenance and network context

The investigation dilemma: Debugging securely



The Scenario

eBPF detects an anomaly
(Unauthorized cudaMemcpy access or a stuck kernel)

The Dilemma

To investigate, operators traditionally rely on
“kubectl exec” or SSH to drop a shell into the running pod

The Risk

- breaks container immutability
- Compromises workload provenance
- Introduces massive security vulnerabilities into highly sensitive AI production environments

Ephemeral Containers: The surgical strike

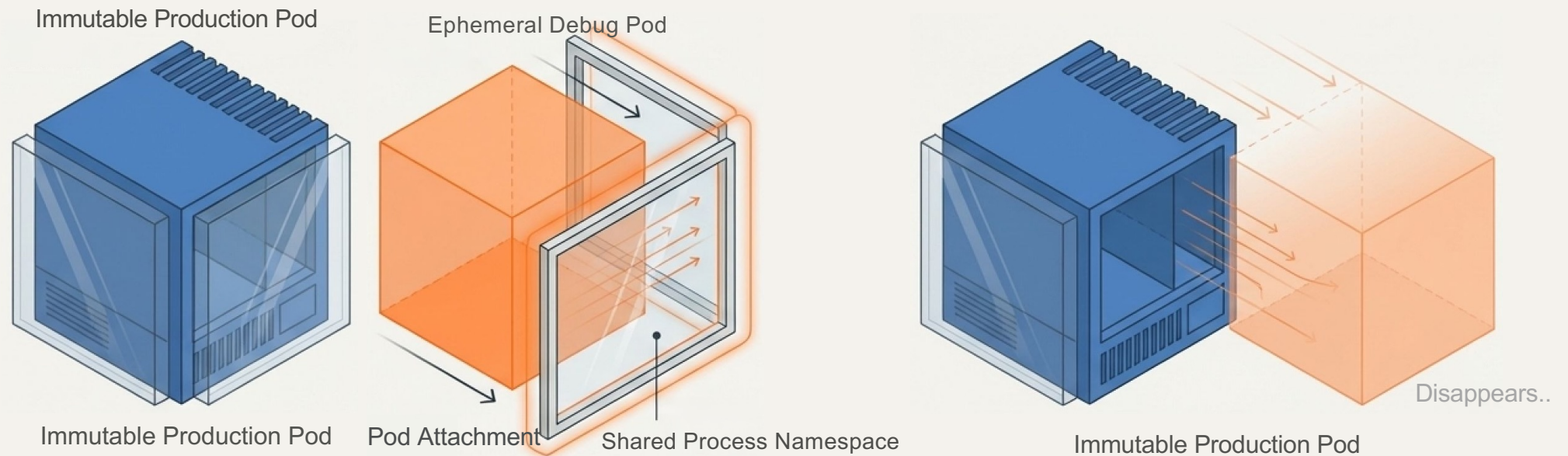


KubeCon



CloudNativeCon

Europe 2026



Dynamic Injection

Inject advanced debugging tools (custom eBPF profilers, network sniffers) directly into a running Pod's namespace

Zero Mutation

The original container image is never altered. No SSH daemons, no bloated base images

Pure Provenance

Guarantees compliance and security while granting deep, real-time investigative access

In Action: The observability hack workflow



KubeCon



CloudNativeCon

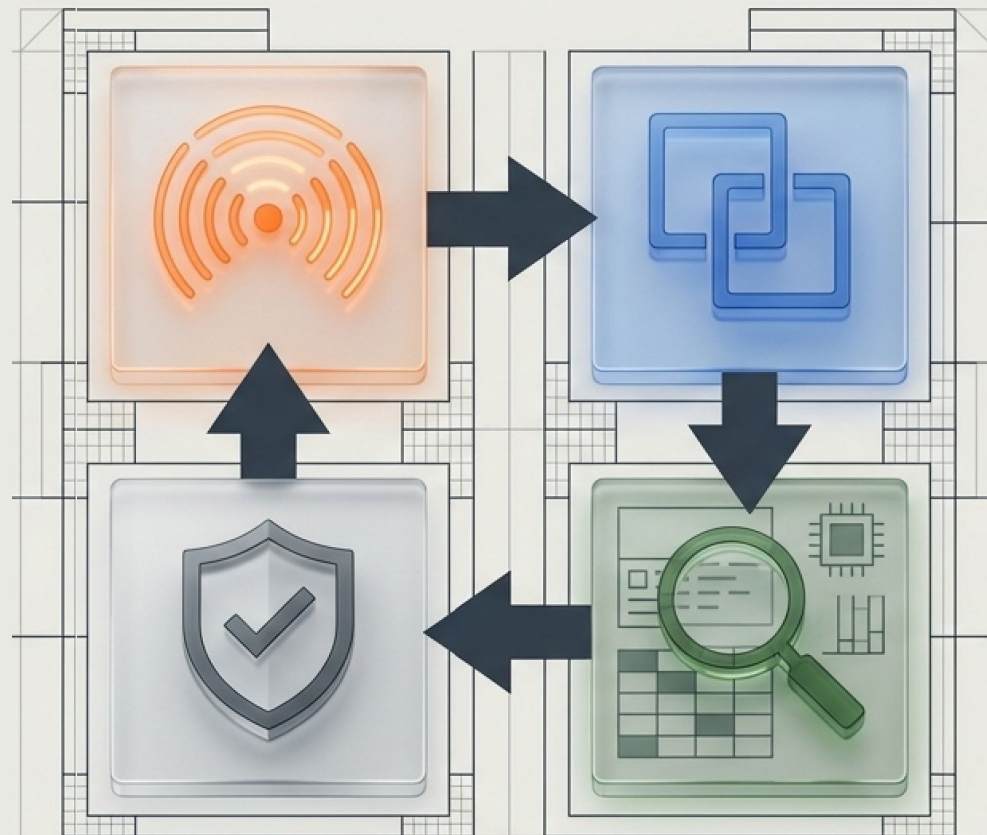
Europe 2026

1. Detect (DCGM + eBPF)

eBPF passively flags anomalous cudaMemcpy (Device-to-Host) activity on the node

4. Remediate (Zero Downtime)

Terminate the specific rogue process and patch the vulnerability without restarting the healthy inference workload



2. Deploy (Ephemeral Container)

Kubernetes operator dynamically launches an Ephemeral Container armed with advanced tracing tools into the target Pod

3. Analyze (Shared Namespace)

Capture the rogue process PID and trace the unauthorized CUDA API calls

The Metrics & Tools

NVIDIA DCGM Exporter

- GPU Utilization
- Memory usage
- Temperature
- MIG partition

eBPF Observability Layer

- + Tools (bcc / libbpf / pixie / parca)
- syscall tracing
- network flow
- process-level activity
- latency breakdown

Runtime Security Layer - Cilium Tetragon

- Detect anomalous process behavior
- Container escape detection
- Detecting unauthorized process execution

Network Observability - Cilium + Hubble

- Pod ↔ Pod traffic
- L7 visibility (HTTP/gRPC)

Visualization / Storage

- Prometheus + Grafana
- OpenTelemetry (optional)

The Full-Stack GPU Observability Architecture

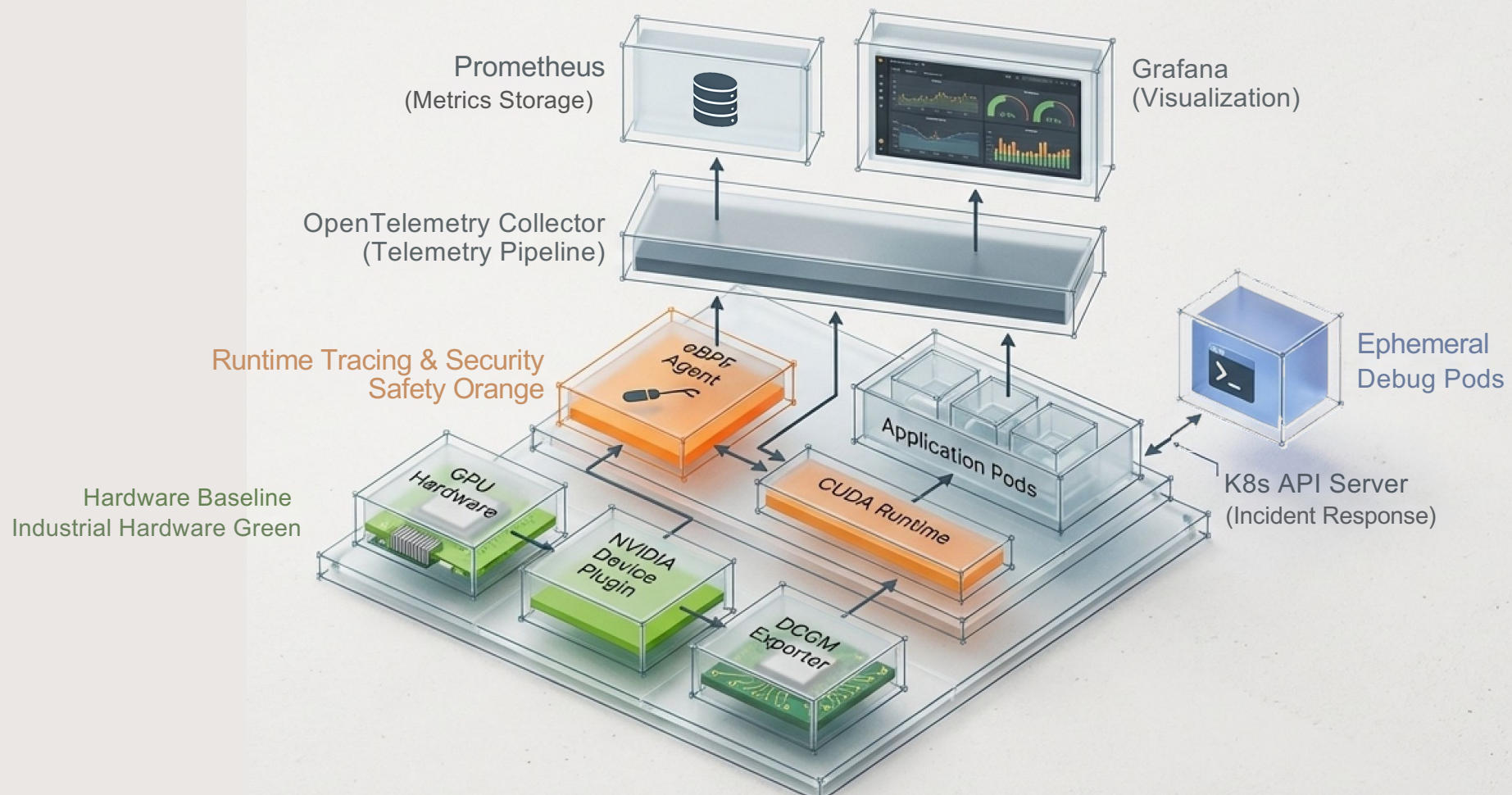


KubeCon



CloudNativeCon

Europe 2026



ML Infrastructure is just Cloud-Native infrastructure

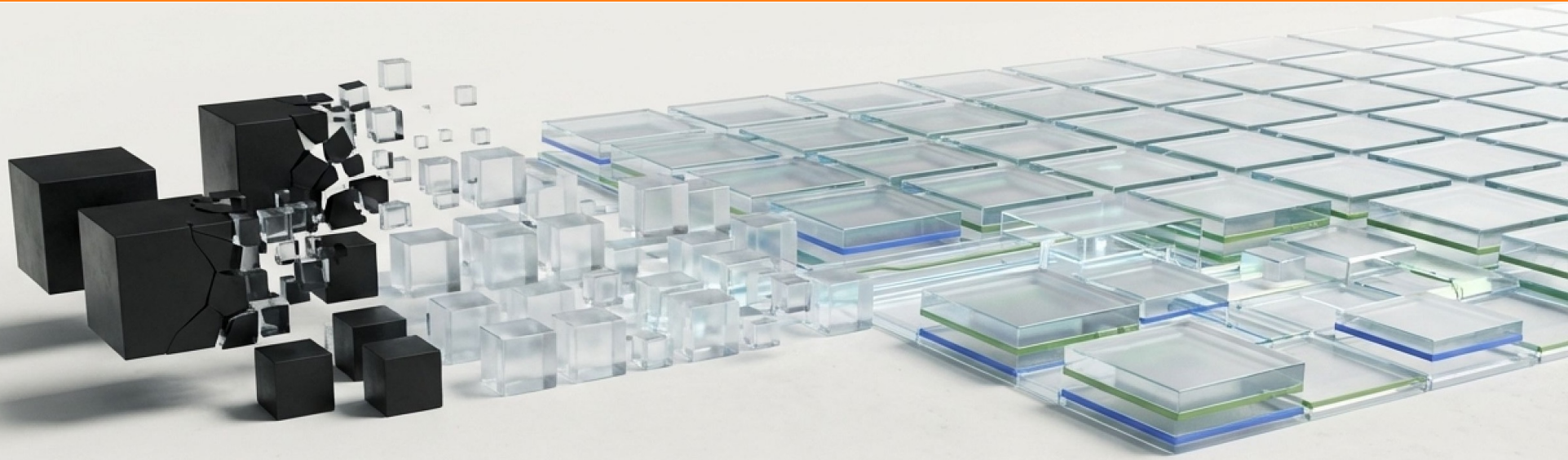


KubeCon

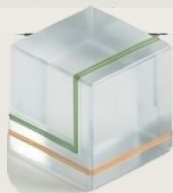


CloudNativeCon

Europe 2026

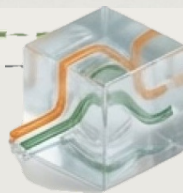


GPUs are No Longer Black Boxes



Hardware metrics alone are a critical blind spot for performance and security

Zero-Cost Visibility



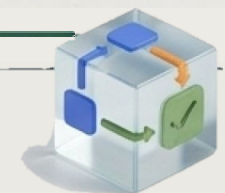
Combine DCGM with eBPF to achieve deep-kernel, application-aware telemetry without altering code

Secure Debugging



Ephemeral containers represent the future of incident response, preserving immutability in high-stakes AI clusters

The End State



Confidently optimize utilization, detect anomalies, and trace the full lifecycle of your GPU workloads

Let's Keep in touch!



KubeCon



CloudNativeCon

Europe 2026



Brandon Kang

Principal Technical Solutions Architect
Akamai Technologies, Cloud Technology Group

<https://www.linkedin.com/in/brandonkang/>

My LinkedIn



Session Slide

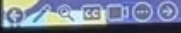




KubeCon CloudNativeCon
Europe 2026

Hacking GPU Observability: eBPF & Ephemeral Containers in Action on Kubernetes

Brandon Kang
Principal Technical Solutions Architect
Akamai Technologies, Cloud Technology Group



Thank You!

