



KubeCon



CloudNativeCon

Europe 2026

#KubeCon #CloudNativeCon

The Missing Half of Performance Profiling: Understanding Memory in Cloud-Native Systems

Dom Delnano

Pixie core maintainer

Gimlet Labs - Member of Technical Staff



Dom Delnano



KubeCon



CloudNativeCon

Europe 2026



[@ddelnano](https://twitter.com/ddelnano)
[@ddelnano.bsky.social](https://bsky.app/profile/ddelnano.bsky.social)

Pixie core maintainer
Member of Technical Staff - Gimlet Labs

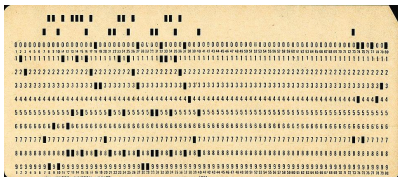
Previous: CrowdStrike, New Relic, Twitter



Computing's Oldest Enemy: Slow programs



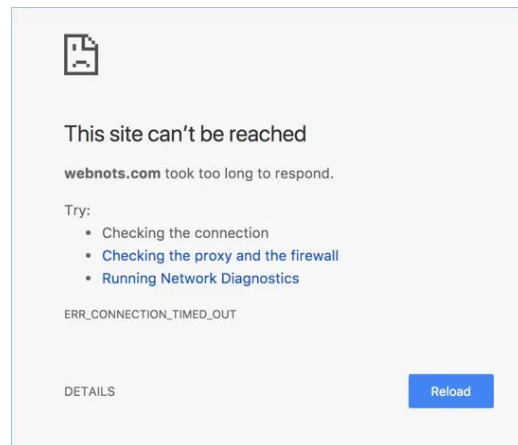
CloudNativeCon
Europe 2026



Mainframe
1960s



Personal Computing
1990s



Modern Cloud
Today

The Evolution of CPU Profiling

Tooling	Scope	Instrumentation	Usage	Overhead
gprof profiling 1980s - 1990s	Single program 	Compile with <code>-pg</code> 	Ad Hoc 	High 
Sampling profilers 2000s - 2010s	Single host 	None 	Ad Hoc 	Low 
Continuous Profilers 2010s - Today	Program, Host or Fleet wide 	None 	Always-on 	Low 

All of Computing's Performance problems solved!

Not Quite ...

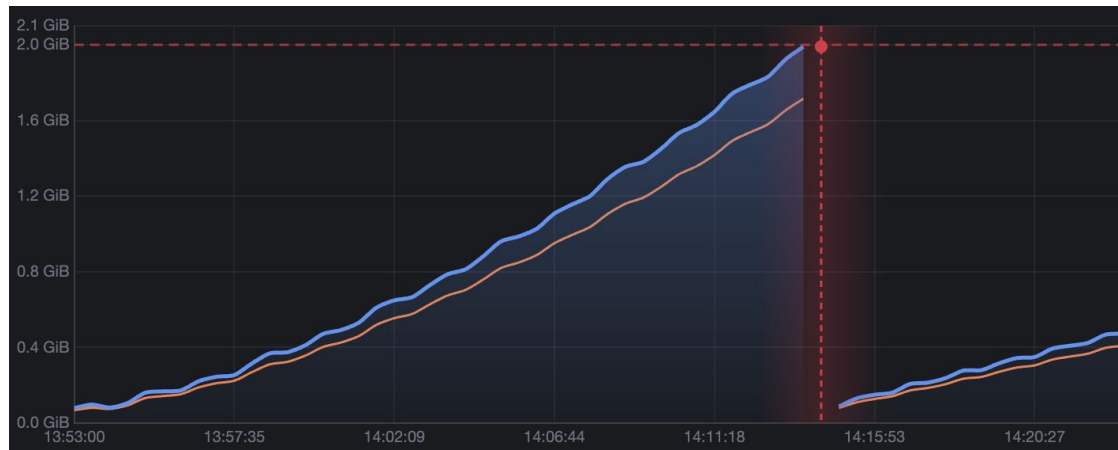


KubeCon



CloudNativeCon

Europe 2026



NAME	READY	STATUS
my-pod-1	0/2	OOMKilled

Not Quite ...



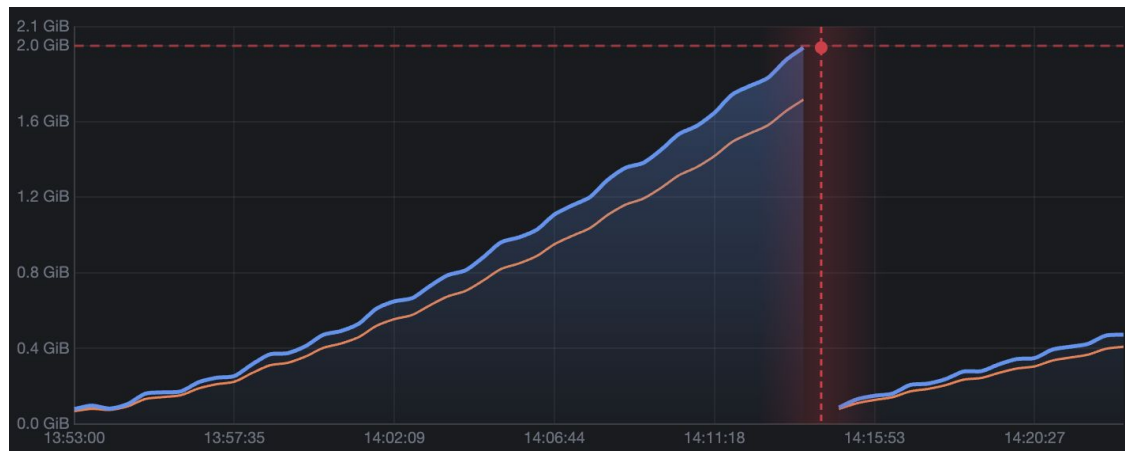
KubeCon



CloudNativeCon

Europe 2026

- Memory leak?
- Working set size > 2GB?



NAME	READY	STATUS
my-pod-1	0/2	OOMKilled

Understanding Memory Use



KubeCon



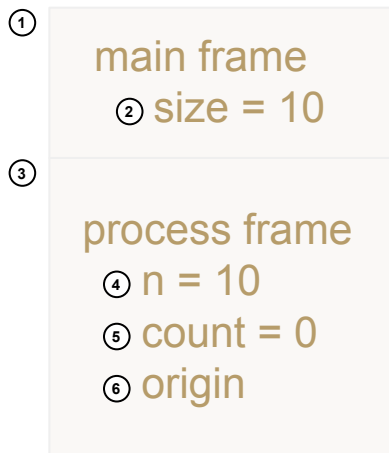
CloudNativeCon

Europe 2026

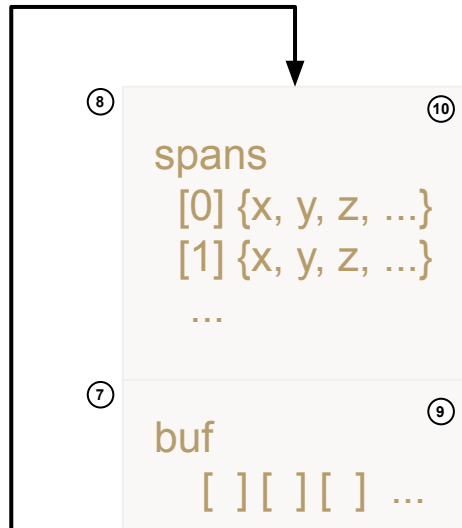
```
1  int main() { ①
2      int size = 10; ②
3
4      process(size); ③
5
6      return 0;
7  }
8
9  void process(int n) { ④
10     int count = 0; ⑤
11     Span origin = {0, 0}; ⑥
12
13     char *buf =
14         malloc(n * sizeof(char)); ⑦
15
16     Span *spans =
17         malloc(n * sizeof(Span)); ⑧
18
19     free(buf); ⑨
20     free(points);
21 }
```

Program

High mem addr



Stack



Low mem addr

Heap

Understanding Memory Use



KubeCon

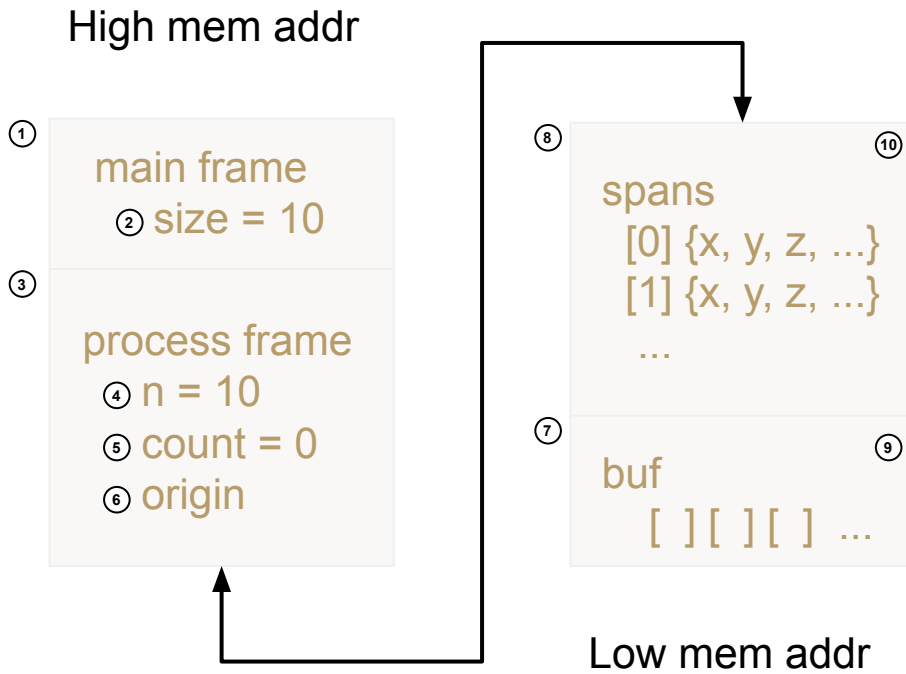


CloudNativeCon

Europe 2026

```
1  int main() { ①
2      int size = 10; ②
3
4      process(size); ③
5
6      return 0;
7  }
8
9  void process(int n) { ④
10     int count = 0; ⑤
11     Span origin = {0, 0}; ⑥
12
13     char *buf =
14         malloc(n * sizeof(char)); ⑦
15
16     Span *spans =
17         malloc(n * sizeof(Span)); ⑧
18
19     free(buf); ⑨
20     free(points); ⑩
21 }
```

Program



Assumption:

Span is 8 bytes

Span[10] = **80 bytes**
(n * sizeof(Span))

char[10] = **10 bytes**
(n * sizeof(char))

Total Allocs: 90 bytes

Total Objects: 20

In-use Allocs: 0 bytes

In-use Objects: 0 bytes

Stack

Heap

Memory Profiling: Allocation Sampling



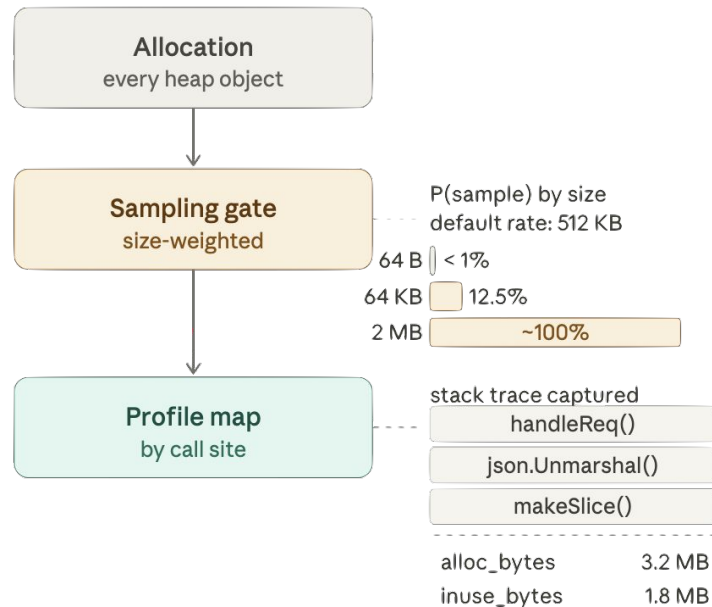
KubeCon



CloudNativeCon

Europe 2026

- Overhead: Low
- Use case: Identifying allocation hot spots
- Temporal: Yes
- Example: Go's pprof, gperftools, tcmalloc



Memory Profiling: In-use Profiling



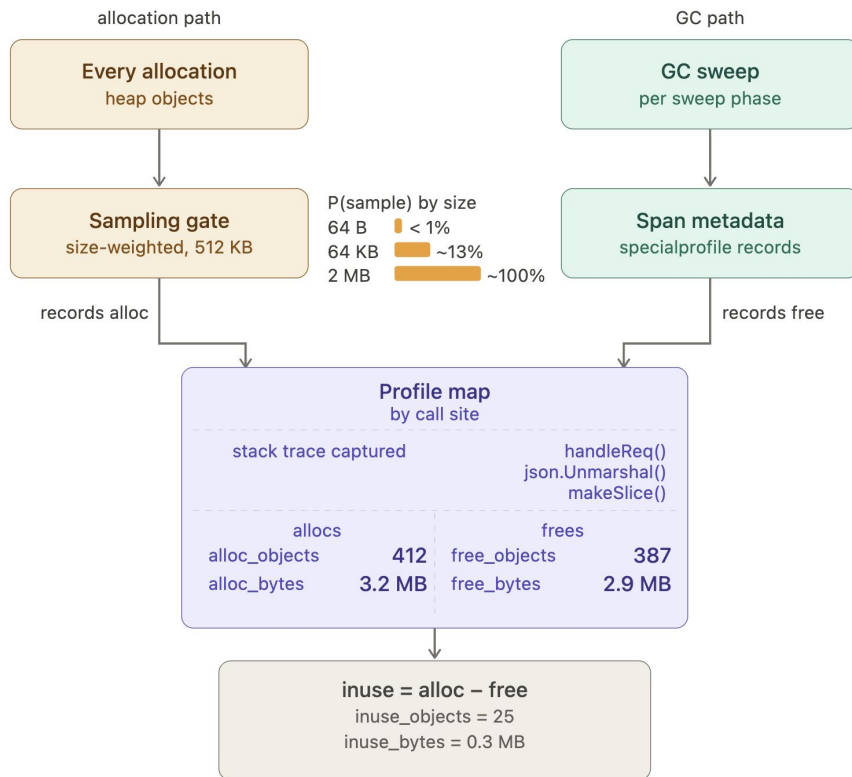
KubeCon



CloudNativeCon

Europe 2026

- Overhead: Low
- Use case: Detecting leaks
- Temporal: Yes
- Example: Go's pprof, gperftools, tcmalloc



Memory Profiling: Full Heap Capture



KubeCon



CloudNativeCon

Europe 2026

- Overhead: High
- Use case: Point in time memory view
- Temporal: No, one-off capture
- Examples: Valgrind, JVM heap dumps, etc.

Memory Profiling: Object Introspection



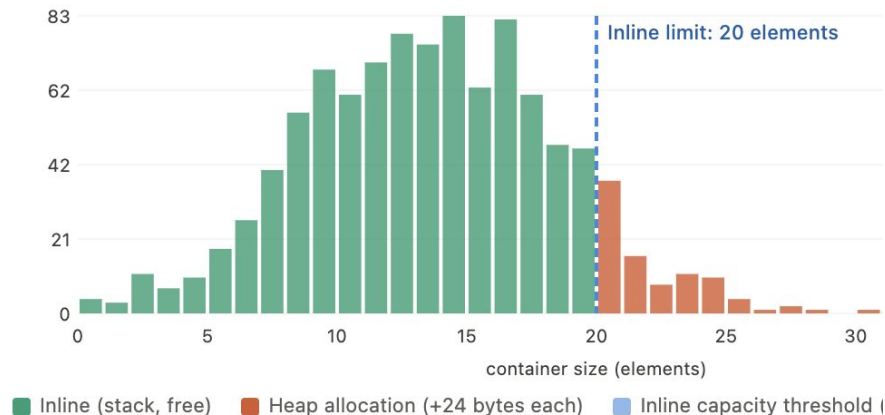
KubeCon



CloudNativeCon

Europe 2026

- Programming language container types have opt. to avoid excess heap allocations
- A container under its static size allocates memory entirely on stack
- Must know dynamic container's runtime distribution or will have worse performance



Distribution of 1,000 `std::vector` objects

Memory Profiling: Object Introspection



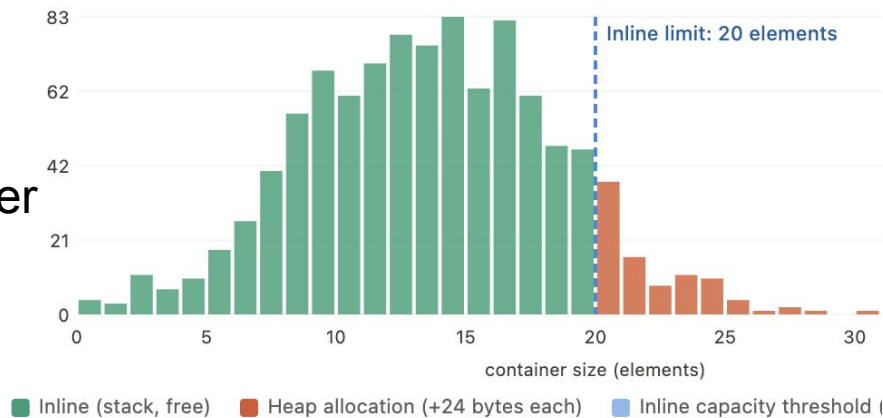
KubeCon



CloudNativeCon

Europe 2026

- Overhead: Medium
- Use case: Avoid heap allocs for container data types
- Examples: [Meta's OI](#)



Container Type	Static Size	Total Heap Allocations (1000 objects dist.)	Heap Bytes
<code>std::vector<int></code>	0	100% heap allocated	<code>sizeof(int) x 1000 instances</code>
<code>small_vector<int, 20></code>	20	9% heap allocated	<code>sizeof(int) x 90 instances</code>

What is Pprof?



KubeCon

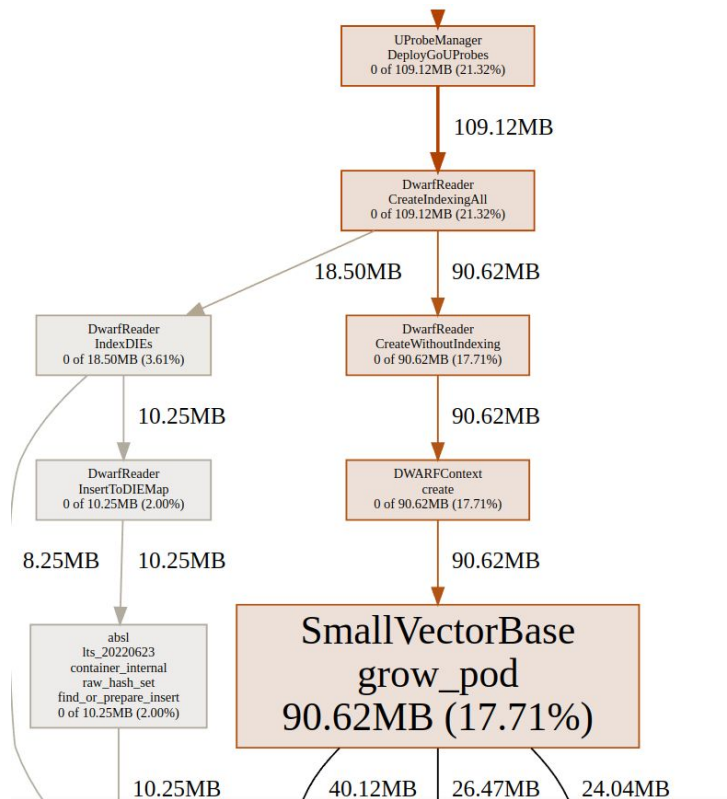


CloudNativeCon

Europe 2026

- Profiling data format (protobuf)
 - Supports CPU and memory profiles
- Size-weighted sampling profiler
- Visualization tool for pprof files

```
1  
2 $ pprof -http=localhost:8888 $heap_profile
```



Memory Profiling: Tool Landscape



KubeCon



CloudNativeCon

Europe 2026

parca



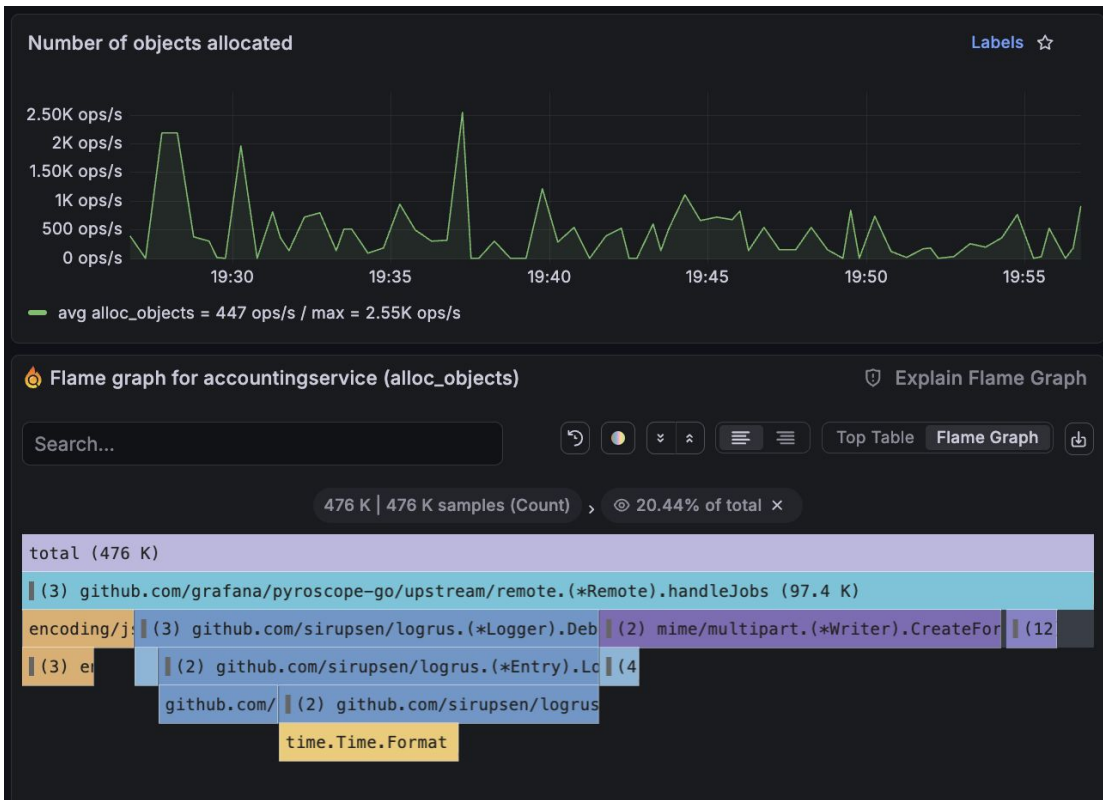
PIXIE

- Profile format: pprof-native ingestion
- Pull model: scrapes `/pprof` endpoints
- Impl: allocation sampling and in-use profiling
- Supports:
 - Go
 - Rust (w/ jemalloc)
 - Python
 - NodeJS

```
1 # parca.yaml
2 scrape_configs:
3   - job_name: "rjem"
4     scrape_interval: "10s"
5     static_configs:
6       - targets: [ '127.0.0.1:3000' ]
7     profiling_config:
8       pprof_config:
9         heap:
10           enabled: true
11           path: /debug/pprof/heap
```

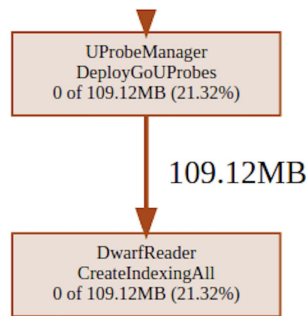
- Profile format: N/A (for push model)
- Push model (primarily) via SDKs
- Impl: allocation sampling and in-use profiling
- Supports:
 - Go
 - Java
 - .NET

```
1 func main() {  
2  
3     pyroscope.Start(pyroscope.Config{  
4  
5         ProfileTypes: []pyroscope.ProfileType{  
6             pyroscope.ProfileAllocObjects,  
7             pyroscope.ProfileAllocSpace,  
8             pyroscope.ProfileInuseObjects,  
9             pyroscope.ProfileInuseSpace,  
10        },  
11    })
```

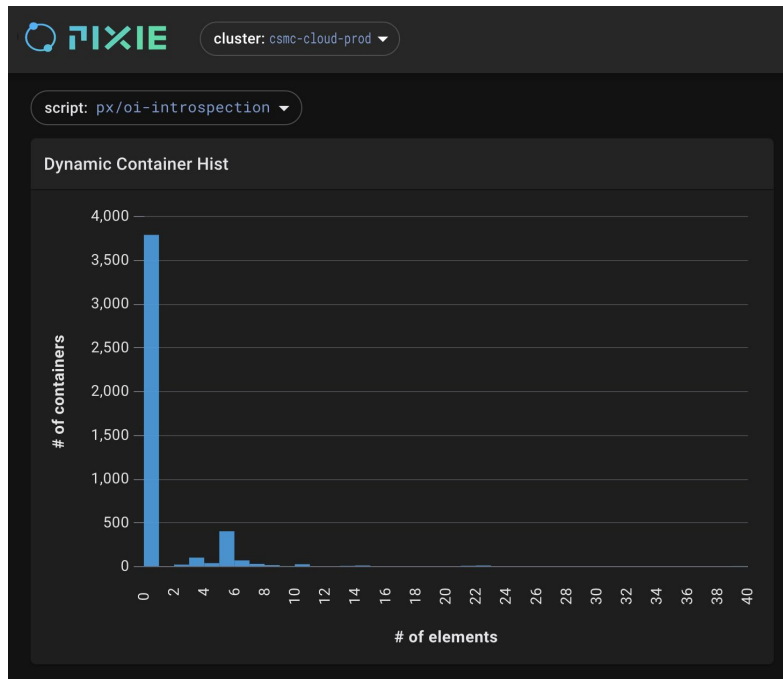


- Active work on memory profiling support
- Planning to support `/pprof` endpoint ingestion
 - Needs integration with Pixie's flame graph viz
- Currently dogfooded to optimize Pixie's own memory footprint

```
1 # Orchestrated via wrapper script (1)
2 $ px run -o json -f collect_heap_dumps.pxl > $pxl_heap_output_file
3
4 $ pprof -http=localhost:8888 $heap_profile
5
```



- Ability to capture object introspection details across a k8s cluster (native binaries)
- Captures runtime distribution of container data types to identify potential memory savings
- Primarily motivated to further optimize Pixie's memory footprint. Available to other native applications to understand their container sizes



Memory Profiling: Comparison



KubeCon



CloudNativeCon

Europe 2026

Technique	Measures	Detects	Mechanism	Overhead	Tools
Allocation sampling	Bytes/objs allocated per call site	Allocation heavy funcs	Size-weighted sampling in runtime	Low	Go pprof, Pyroscope, Parca
In-use profiling	Live bytes per call site	Memory leaks	Same as above	Low	Go pprof, Pyroscope, Parca
Full heap capture	All live objects, stop-the-world snapshot	Detailed point in time memory	Runtime pause + dump	High	Valgrind, heaptrack, etc.
Object introspection	Stack & heap size of dyn. containers	Dynamic container distribution	Custom traversal at runtime	Medium	Facebook OI, Pixie

Performance Profiling Recap



KubeCon



CloudNativeCon

Europe 2026

- CPU Profiling: Mature and easy to adopt with available tooling
- Prefer sampling based memory tools in production
 - Size-based sampling lowers overhead while still identifying problematic allocations
 - pprof standardized CPU and memory profiles format
- Broader ecosystem
 - Too fragmented due to language-specific requirements
 - Most tooling has too much overhead

Future of Memory is Tiered



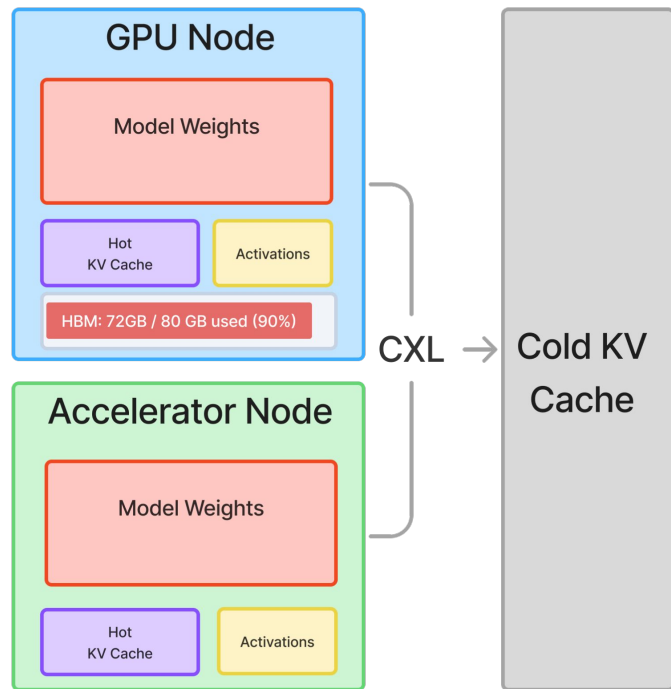
KubeCon



CloudNativeCon

Europe 2026

- Cloud apps largely use host (CPU) memory
- AI apps have unique memory requirements
 - Uses host and device memory
 - Weights and KV caches dwarf heap sizes
 - RDMA the new RPC layer
- Tiered-memory will be commonplace
 - Host (CPU) memory
 - GPU Memory
 - Accelerator SRAM (Groq, Cerebras)
 - CXL: cross-host memory access



Thank you!



KubeCon



CloudNativeCon

Europe 2026

