

Deep Application/OS Insight via eBPF and Zabbix

Karel Bělunek · TSP data · Zabbix Summit 2026, Riga



The application is slow. Monitoring says everything is fine.

CPU fine, RAM fine, disk fine, ping fine. Yet the application randomly loses performance and availability. Users see it, dashboards don't.

Ghost hunting

You know something is there. You have nothing to measure it with.

System state $\neq$ application experience

Classic metrics tell you how the server feels. Not how the application feels.

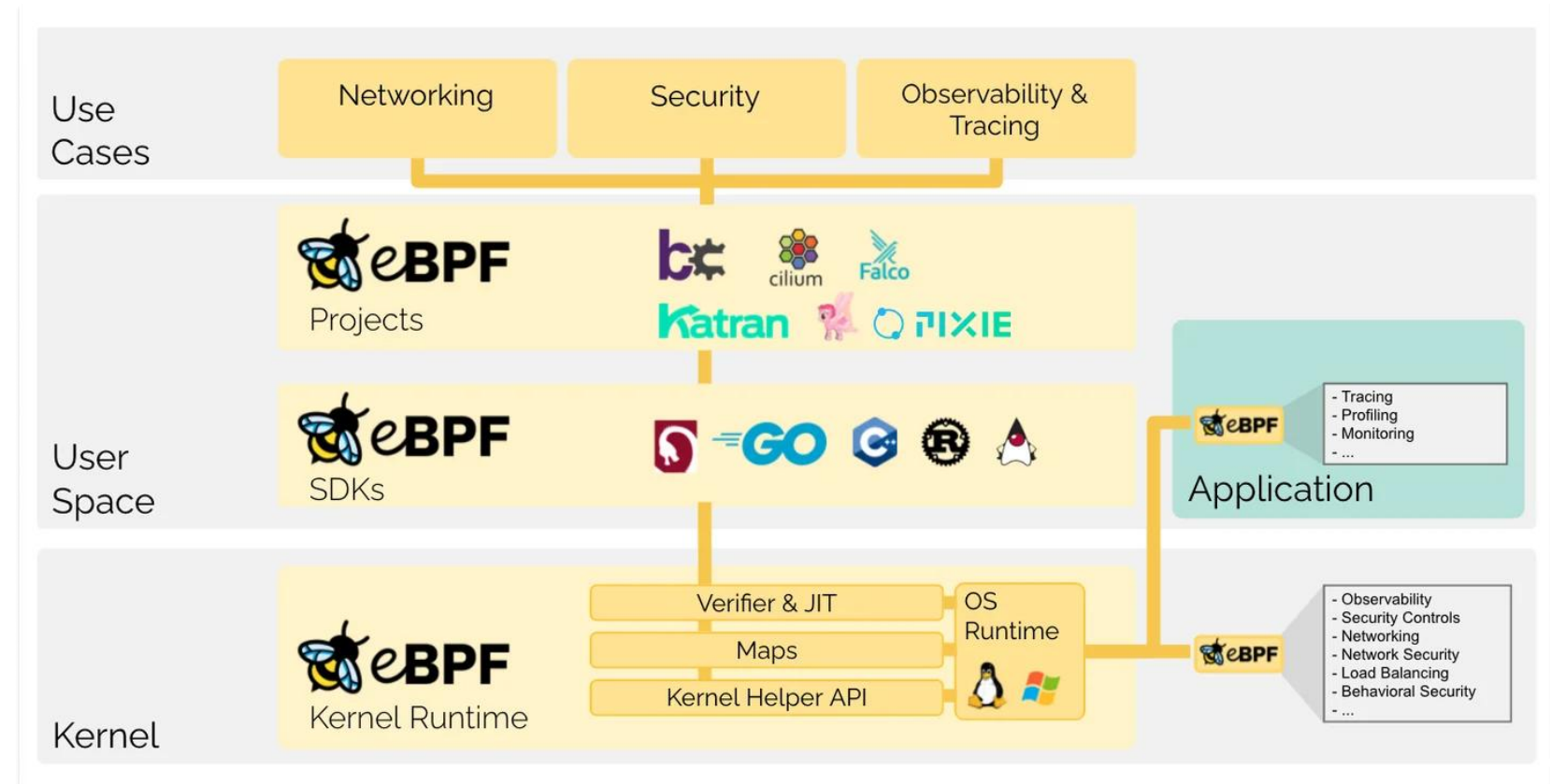
→ Deeper insight into applications and the OS

One way to get there: eBPF tracing.



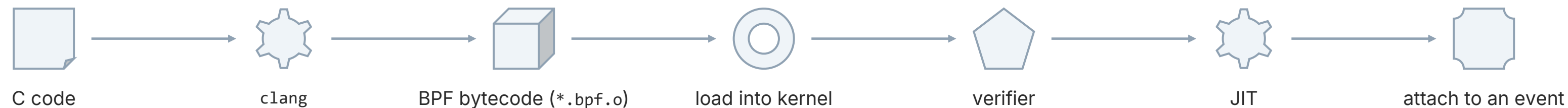
What is eBPF

- **extended Berkeley Packet Filter** — not from Berkeley, not about packets, not a filter. *Three words, three lies.*
- In reality: a **sandboxed VM inside the Linux kernel**. It runs your verified code in response to an event in the kernel or in an application.
- Alexei Starovoitov, 2014 (kernel 3.18). Today: **networking · security · observability**.
- Hyperscalers have run it in production for years: Kubernetes Dataplane V2 (Cilium), Android, Meta, Netflix.
- From the Zabbix perspective: the same game changer as **JavaScript preprocessing (Duktape)**.
- <https://ebpf.io/what-is-ebpf/>

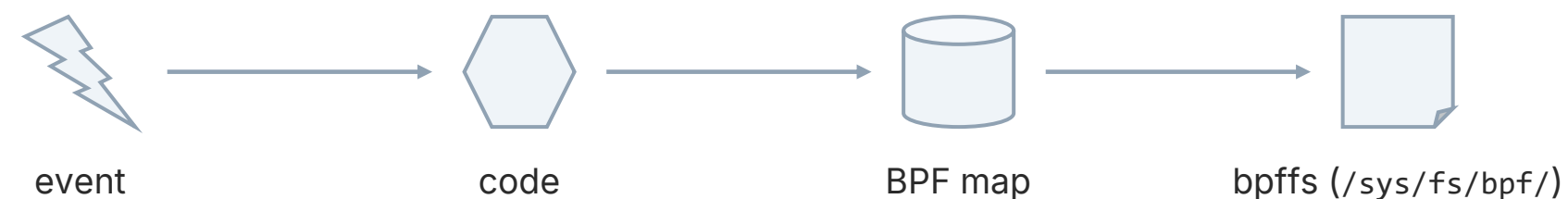


How eBPF works

Build time — once



Runtime — on every event



kprobe / fentry

Entry / exit of a kernel function.

tracepoint

Stable, documented points in the kernel: scheduler, syscalls, TCP...

uprobe / uretprobe

Entry / exit of a userspace function: getaddrinfo(), SSL_do_handshake().

Try this at home

```
bpftool prog list    # which eBPF programs are already running on your Linux box
```

Kernel → userspace: why counters, not events

Classic tracing

Every event → ring buffer → userspace → parsing → ship it somewhere. 1,000 `getaddrinfo()` calls/s = 1,000 events/s. **The measurement starts to affect what it measures.**

Our approach: in-kernel aggregation

Simple code providing:

- Counter: `map[key]++`
- Histogram: `map[bucket(latency)]++`

No event ever leaves the kernel.

Userspace reads a handful of numbers, once a minute.

What this deliberately is NOT

Distributed tracing. Packet inspection. There are specialised tools for that — and they need a lot of RAM.

DNS histogram: 17 numbers

DNS example: 17 counters = how many `getaddrinfo()` calls took $\leq 100\ \mu\text{s}$, $\leq 1\ \text{ms}$, $\leq 10\ \text{ms}$... $> 30\ \text{s}$. One call = one ++ in one bucket.

The entire “complex” eBPF code. Simplified, but not much.

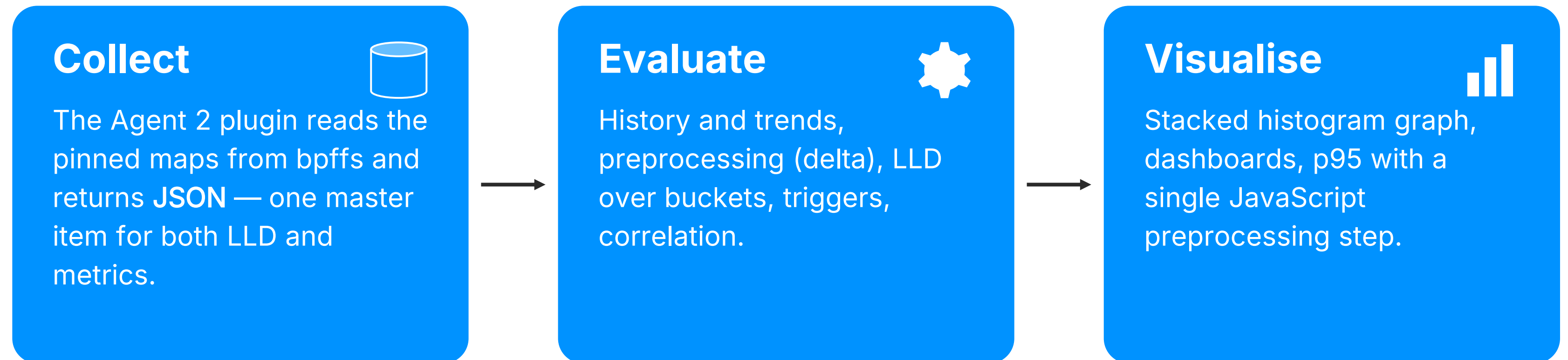
```
C dns_getaddrinfo.bpf.c U ●
examples > dns_latency > C dns_getaddrinfo.bpf.c
1  /* 17 buckets, each = number of calls with that duration (pinned to bpffs) */
2  struct { ... __uint(max_entries, 17); } hist SEC(".maps");
3  /* in-flight calls: tid → entry timestamp (internal, not pinned) */
4  struct { ... } starts SEC(".maps");
5  ✨
6  SEC("uprobe//lib64/libc.so.6:getaddrinfo") /* function ENTRY */
7  int on_enter(void *ctx)
8  {
9      u64 tid = bpf_get_current_pid_tgid();
10     u64 now = bpf_ktime_get_ns();
11     bpf_map_update_elem(&starts, &tid, &now, BPF_ANY); /* remember the time */
12     return 0;
13 }
14
15 SEC("uretprobe//lib64/libc.so.6:getaddrinfo") /* function RETURN */
16 int on_return(void *ctx)
17 {
18     u64 tid = bpf_get_current_pid_tgid();
19     u64 *start = bpf_map_lookup_elem(&starts, &tid);
20     if (!start) return 0;
21     u64 delta = bpf_ktime_get_ns() - *start; /* how long it took */
22     u32 bucket = latency_bucket(delta); /* ≤100 μs → 0 ... >30 s → 16 */
23     u64 *cnt = bpf_map_lookup_elem(&hist, &bucket);
24     if (cnt) __sync_fetch_and_add(cnt, 1); /* +1 in exactly one bucket */
25     bpf_map_delete_elem(&starts, &tid);
26     return 0;
27 }
```

No event, no buffer, no copying. A single ++.

...and here comes Zabbix

There are 17 numbers sitting in a file in the kernel. Who will collect them, watch them and draw them?

eBPF measures. Zabbix does everything else. Business as usual.



Our implementation: two small Go utilities, GPL-3, github.com/tsp-data

zabbix-ebpf-loader

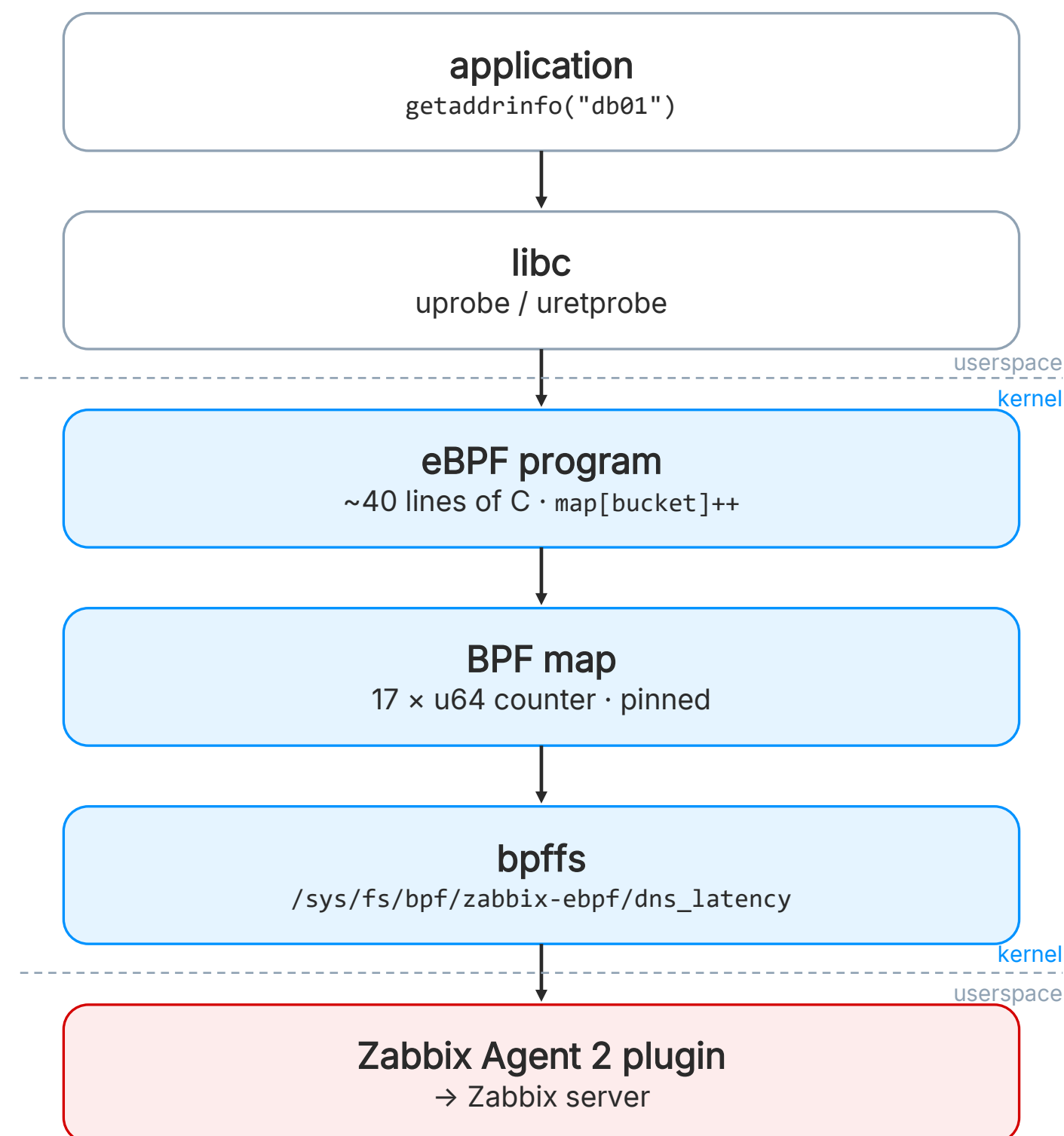
root · systemd service

- loads *.bpf.o, attaches programs (tracepoint / kprobe / uprobe...)
- pins selected maps to bpffs

zabbix-agent2-plugin-bpffs

user zabbix

- loadable Agent 2 plugin — only reads pinned maps
- zbx_ebpf.get → JSON of all metrics · zbx_ebpf.metric[name] → a single value



Processing in Zabbix — business as usual

2 · LLD

Dependent discovery rule,
JavaScript preprocessing: flat JSON
→ {#BUCKET}, {#BUCKET_LABEL}.

4 · Preprocessing

Change per second — a monotonic
counter becomes a rate of calls/s.

1 · Master item

zbx_ebpf.get — one master item
JSON per minute.

3 · Dependent items

Item prototype per bucket,
JSONPath to the ...bucket[{#BUCKET}]
key in the master JSON.

5 · Triggers + graph

"bucket ≥ 2 s growing for 3 checks
in a row". Stacked graph = the
shape of latency over time. Bonus:
p95 via JavaScript.

The case of the stalling collector — a detective story from a real project

Clue 1

DNS is suspected. But measuring the resolution time of one specific hostname — **always OK**.

Clue 3

A second eBPF module collects the **hostnames of the slow queries**. A few days later: a sample of several hundred names.

Crime scene

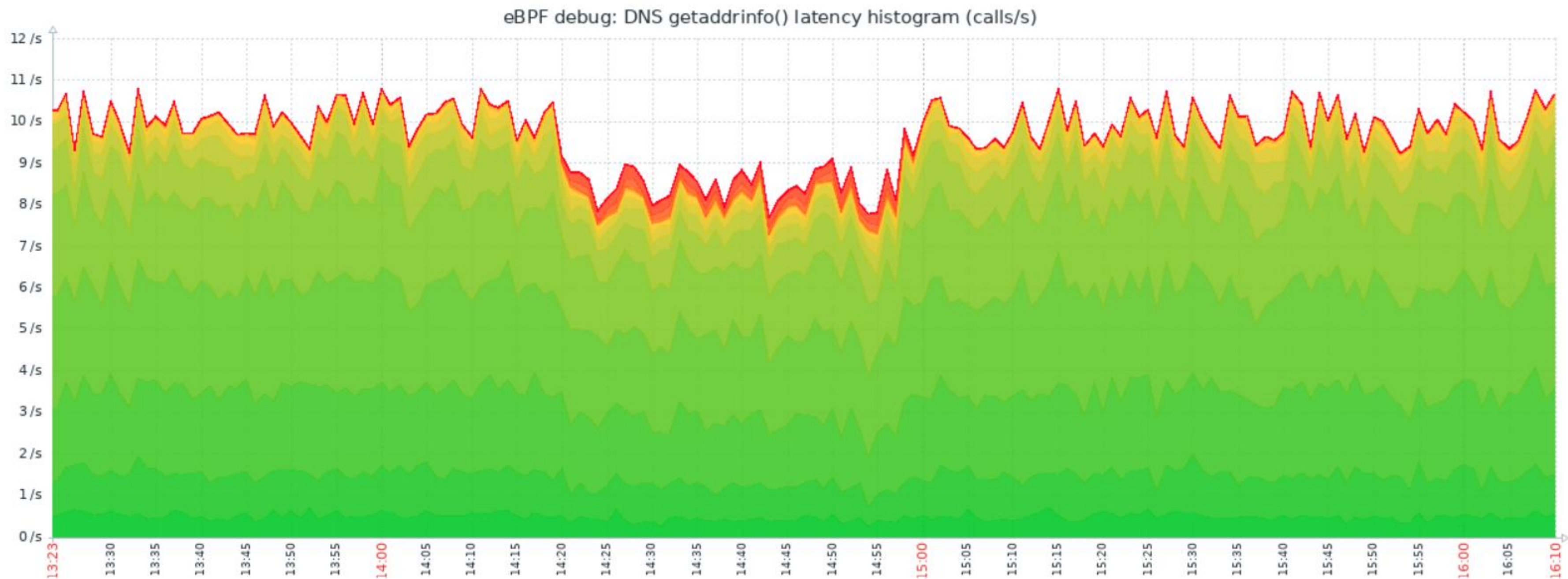
Customer (an ISP), we support a legacy commercial monitoring tool. The metric collector **stalls for no apparent reason** — pollers miss their poll period. It comes randomly and goes away by itself.

Clue 2

eBPF `getaddrinfo()` histogram: now and then the number of **slow** resolutions grows. Not all of them — always just **a few percent**.

Revelation

The customer had acquired a smaller ISP. Names of its devices are forwarded to a **different DNS backend** — and that one had a problem.



Clue 2 — a few percent of calls in the 1–5 s band is enough for the collector to miss its poll period. The classic DNS check stayed green the whole time.

Another ghost, different layer: a REST API and an invisible security scanner

HTTPS REST API “occasionally slow”. /health green, SQL on the DB backend still as fast as ever, network OK. At first it was not at all obvious what was causing the delay.

Clue: only large responses were delayed

Found with eBPF: uprobes on SSL_read / SSL_write in the application, a histogram of latency. /health returns just OK — always fast.

Revelation: an inline security scanner

We later learned that security had deployed a content scanner. Scanning a large payload = seconds added. The health check measured the path, not the work.

Synthetic check vs. eBPF

A check measures what the application could experience. eBPF measures what it **does** experience.

Build your own: three prompts, three spells

Writing eBPF code and finding the right event are both hard and require non-trivial knowledge. An LLM helps with both.

1 · Study

"There is a repo at github.com/tsp-data showing how to connect Zabbix and eBPF. Study it."

2 · Describe the ghost

"I have this ghost-hunting problem. Can it be debugged with this approach? Which eBPF events could I use?"

3 · Implement

"The design looks good. Implement it as a new module."

```
nm -D /lib64/libc.so.6 | grep getaddrinfo  
go tool nm ./app | grep -i 'tls.*Handshake'  
bpftrace -l 'tracepoint:sched:*
```

```
# symbols in a shared library  
# symbols in a Go binary  
# what the kernel offers
```

Takeaway — please wake up your neighbours. This is the part I need you to remember.

1. **eBPF** = a sandboxed VM in the Linux kernel. It runs your code in response to an event — in the kernel and in the application. No modules, no patches, no restart.
2. It enables monitoring **from the application's point of view** — it measures the real `getaddrinfo()`, the real TLS handshake. Not the state of the system around it.
3. Full tracing = load. **We only collect counters and histograms**, aggregated in the kernel, exposed as files in `bpffs`.
4. **Zabbix** reads `bpffs` and does what it does best: LLD, history, preprocessing, triggers, trends. Business as usual.
5. A new module = **40 lines of C**. Not trivial — an LLM will help.



Questions?

github.com/tsp-data/zabbix-ebpf-loader

github.com/tsp-data/zabbix-agent2-plugin-bpffs

ebpf.io — What is eBPF?

Karel Bělunek · TSP data · karel.belunek@tspdata.cz

