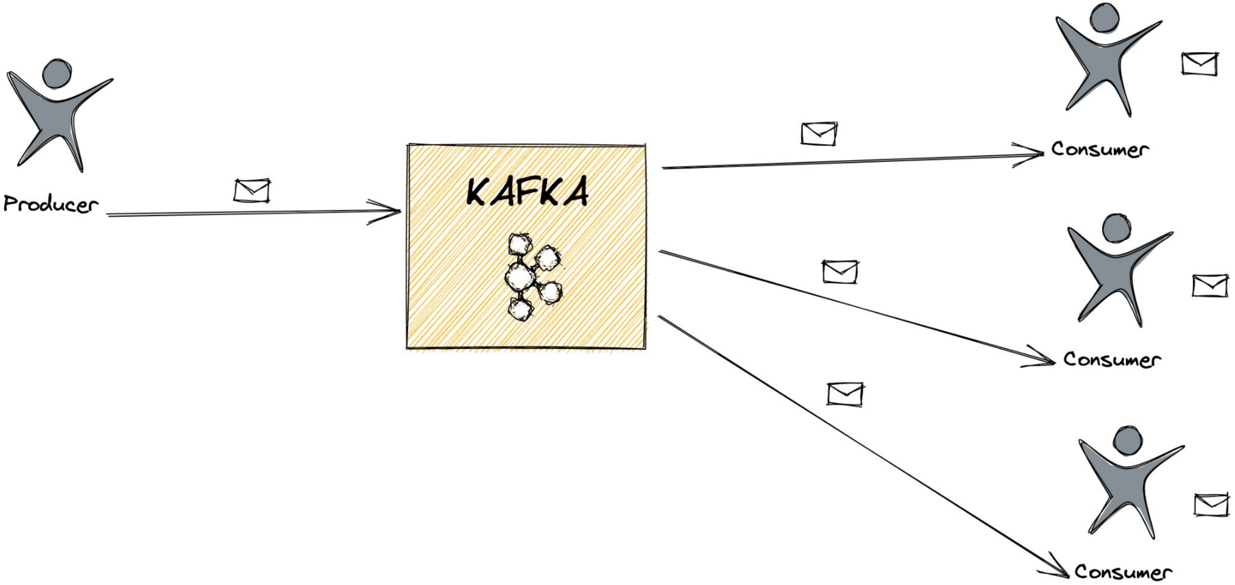
Monitoring Kafka Without Instrumentation Using eBPF

Ryan Cheng & Anton Rodriguez

DEVCONF.cz

Our experience with Kafka

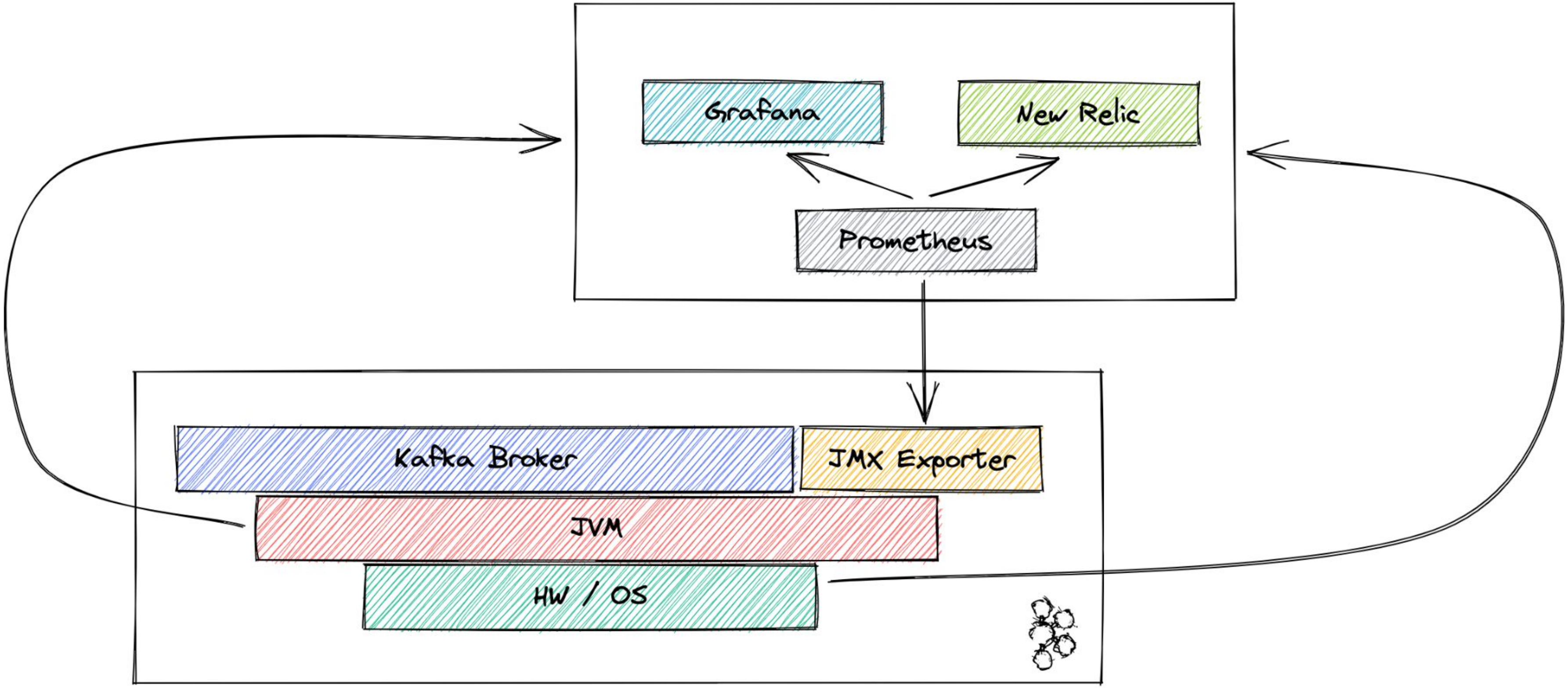


125 PiB of data per month

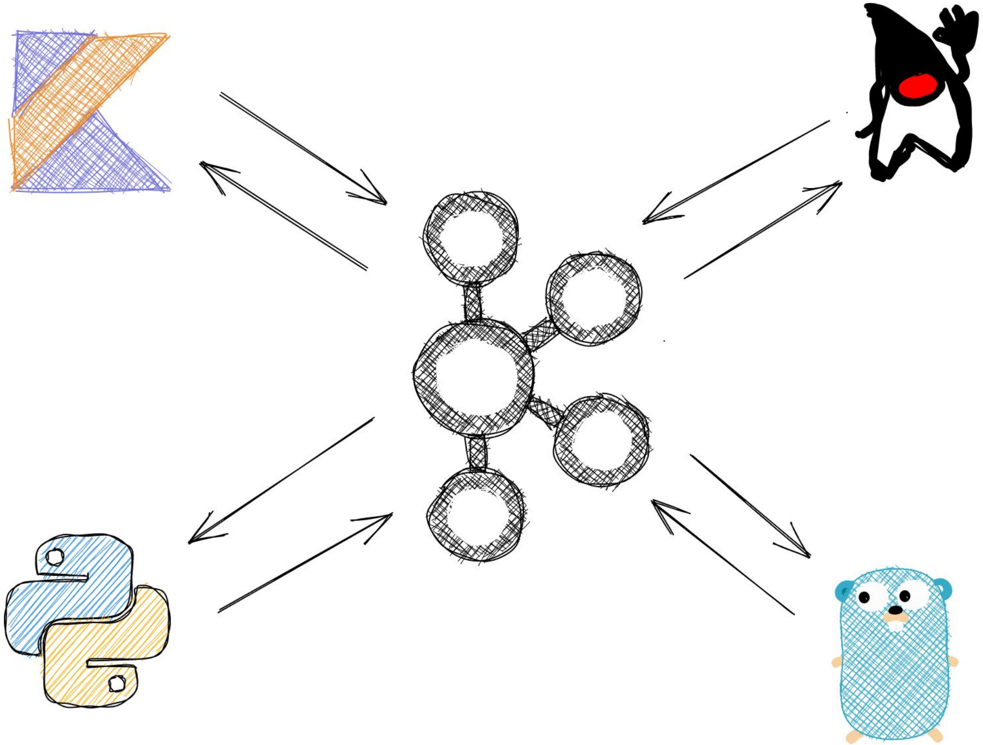
3 B data points per minute

production cluster of **~275 brokers** was running at **20 GB/sec**

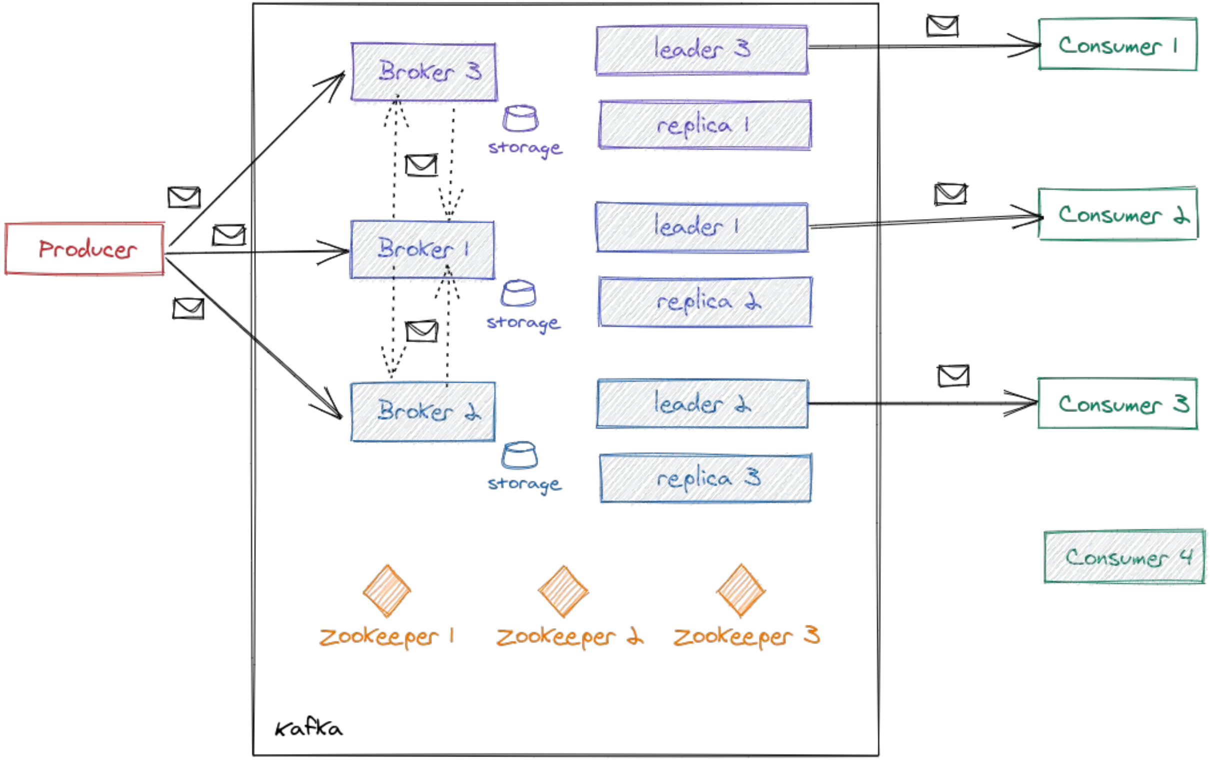
Kafka & observability



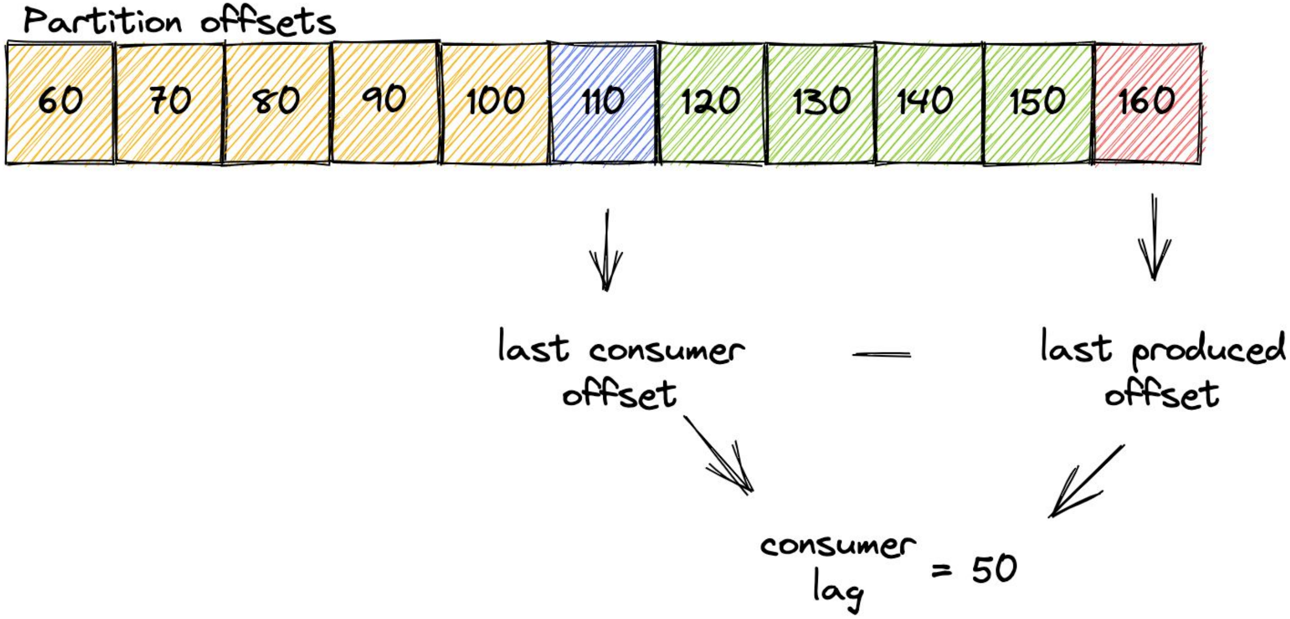
Kafka ecosystem



Kafka rebalances



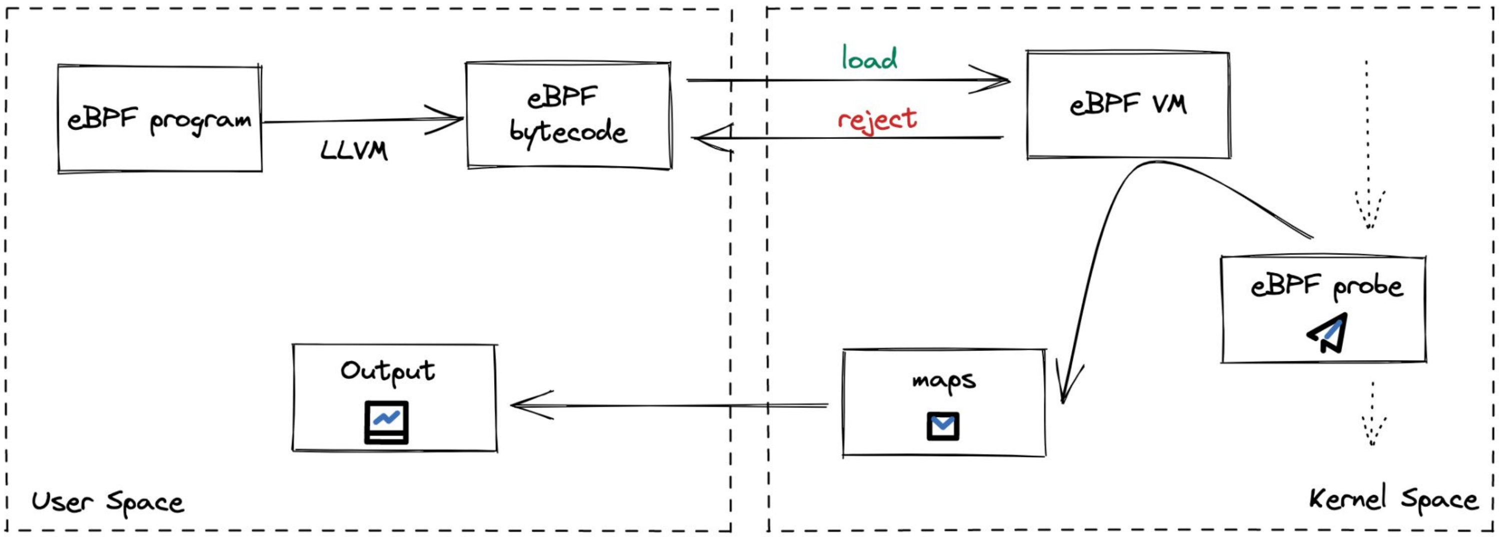
Kafka Consumer Lag



https://github.com/linkedin/Burrow https://github.com/lightbend/kafka-lag-exporter

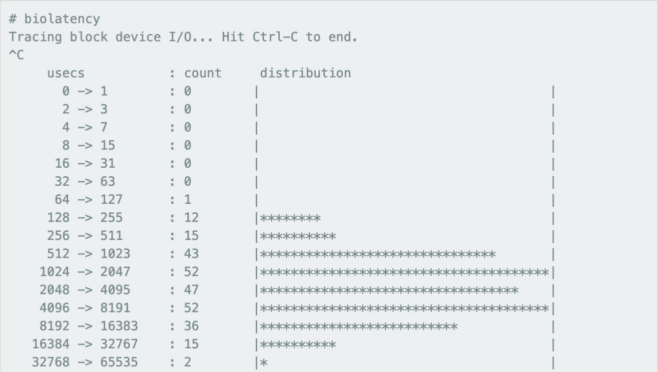
eBPF

eBPF programs (user-defined, sandboxed bytecode executed by the kernel) allow user-defined instrumentation on a live kernel image that can never crash, hang or interfere with the kernel negatively



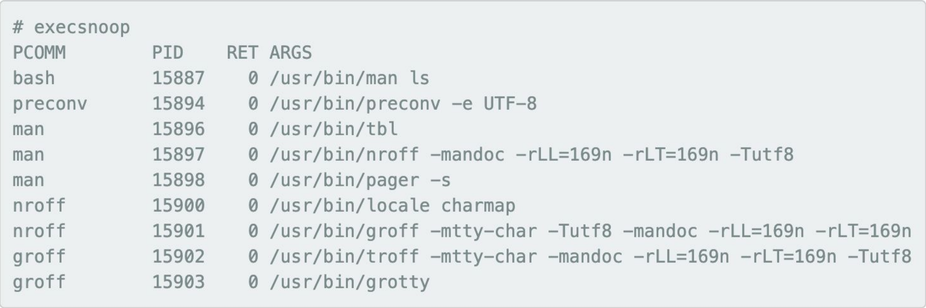
eBPF & BCC

DISK I/O LATENCY HISTOGRAM



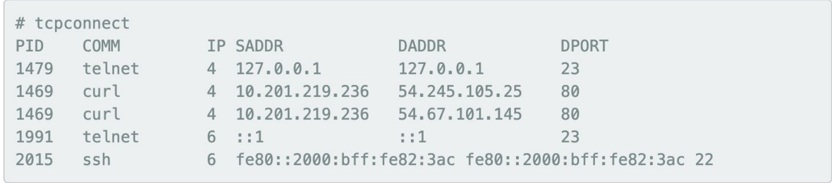
```
# biolatency
Tracing block device I/O... Hit Ctrl-C to end.
^C
     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 1        |                                        |
       128 -> 255        : 12       |*********                               |
       256 -> 511        : 15       |***********                             |
       512 -> 1023       : 43       |*********************************       |
      1024 -> 2047       : 52       |****************************************|
      2048 -> 4095       : 47       |************************************    |
      4096 -> 8191       : 52       |****************************************|
      8192 -> 16383      : 36       |***************************             |
     16384 -> 32767      : 15       |***********                             |
     32768 -> 65535      : 2        |*                                       |
```

NEW PROCESSES



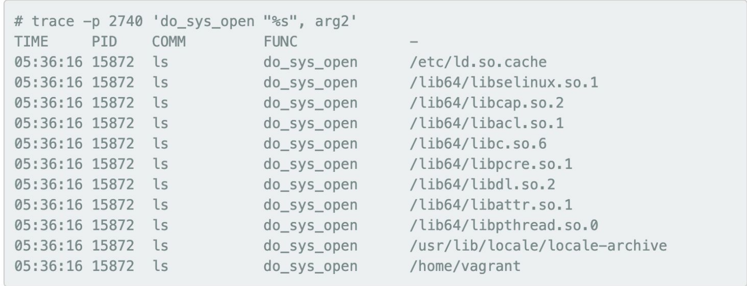
```
# execsnoop
PCOMM            PID    RET ARGS
bash             15887    0 /usr/bin/man ls
preconv          15894    0 /usr/bin/preconv -e UTF-8
man              15896    0 /usr/bin/tbl
man              15897    0 /usr/bin/nroff -mandoc -rLL=169n -rLT=169n -Tutf8
man              15898    0 /usr/bin/pager -s
nroff            15900    0 /usr/bin/locale charmap
nroff            15901    0 /usr/bin/groff -mtty-char -Tutf8 -mandoc -rLL=169n -rLT=169n
groff            15902    0 /usr/bin/troff -mtty-char -mandoc -rLL=169n -rLT=169n -Tutf8
groff            15903    0 /usr/bin/grotty
```

TCP CONNECTIONS



```
# tcpconnect
PID    COMM         IP SADDR            DADDR            DPORT
1479   telnet       4  127.0.0.1        127.0.0.1        23
1469   curl         4  10.201.219.236   54.245.105.25    80
1469   curl         4  10.201.219.236   54.67.101.145    80
1991   telnet       6  ::1              ::1              23
2015   ssh          6  fe80::2000:bff:fe82:3ac fe80::2000:bff:fe82:3ac 22
```

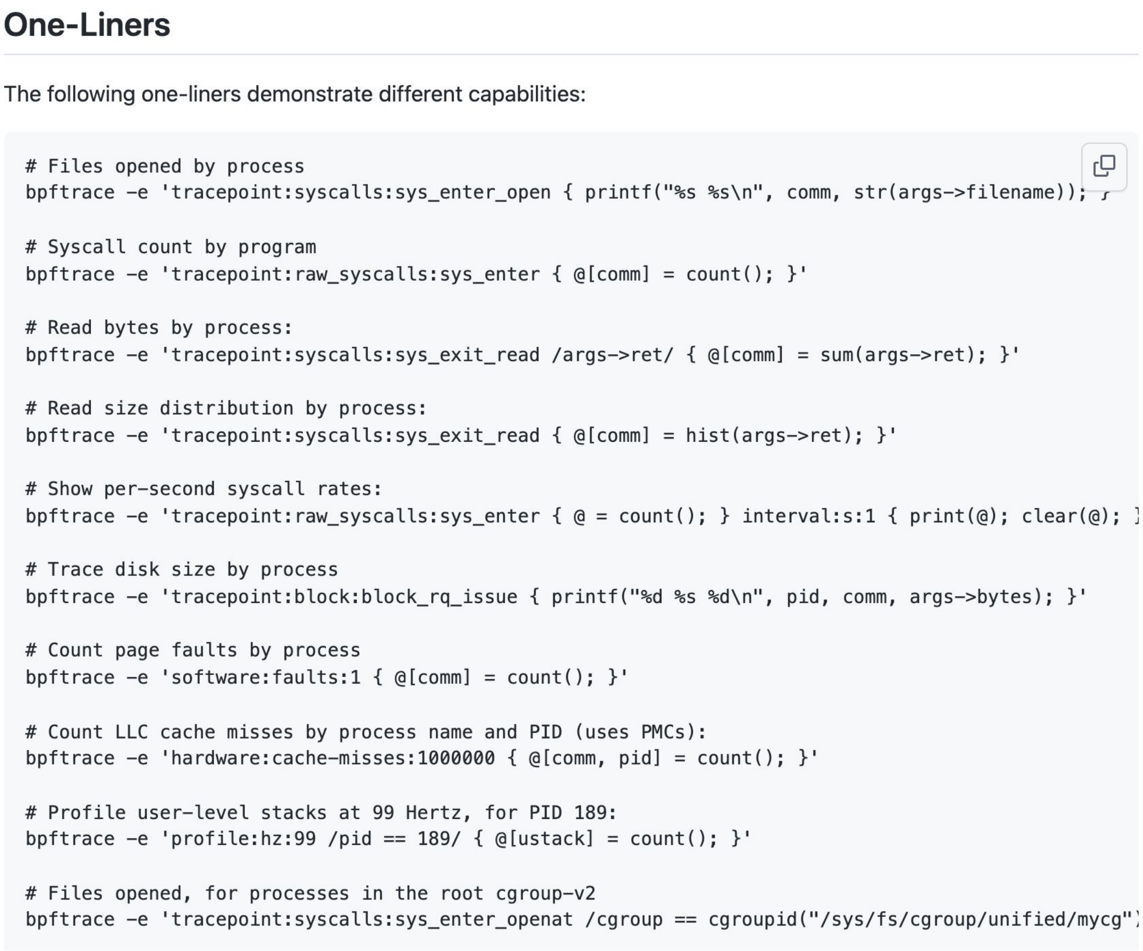
CUSTOM TRACING



```
# trace -p 2740 'do_sys_open "%s", arg2'
TIME     PID    COMM         FUNC             -
05:36:16 15872  ls           do_sys_open      /etc/ld.so.cache
05:36:16 15872  ls           do_sys_open      /lib64/libselinux.so.1
05:36:16 15872  ls           do_sys_open      /lib64/libcap.so.2
05:36:16 15872  ls           do_sys_open      /lib64/libacl.so.1
05:36:16 15872  ls           do_sys_open      /lib64/libc.so.6
05:36:16 15872  ls           do_sys_open      /lib64/libpcre.so.1
05:36:16 15872  ls           do_sys_open      /lib64/libdl.so.2
05:36:16 15872  ls           do_sys_open      /lib64/libattr.so.1
05:36:16 15872  ls           do_sys_open      /lib64/libpthread.so.0
05:36:16 15872  ls           do_sys_open      /usr/lib/locale/locale-archive
05:36:16 15872  ls           do_sys_open      /home/vagrant
```

https://iovisor.github.io/bcc/

bpftrace
kubectl-trace



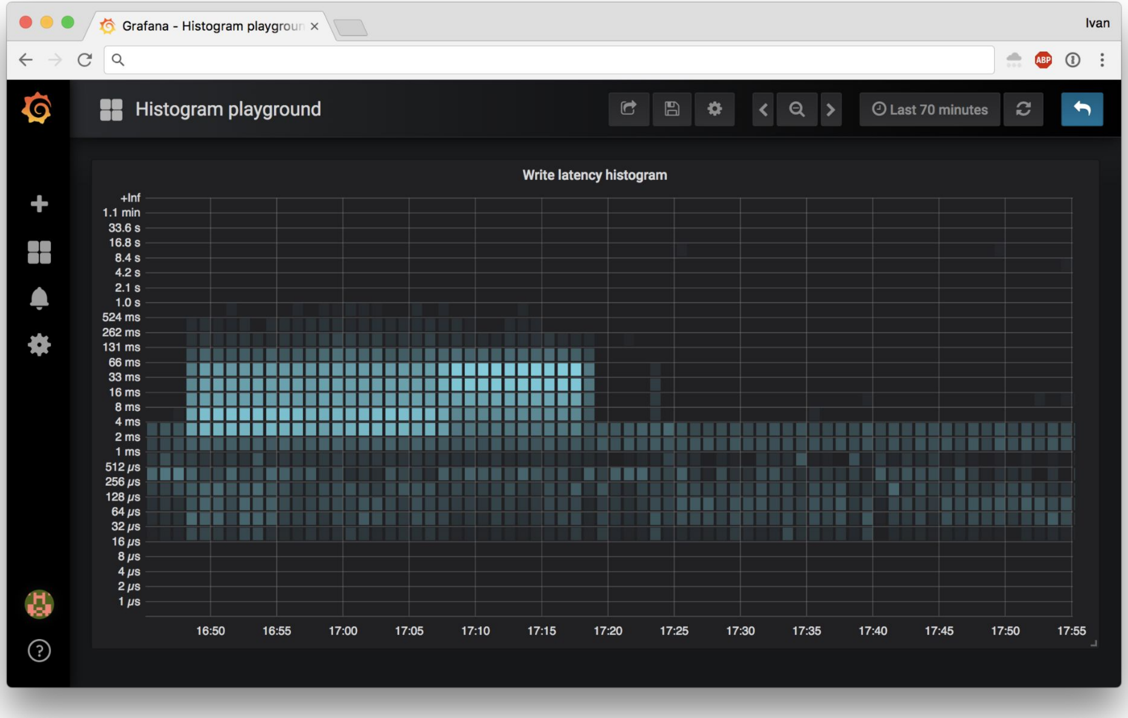
One-Liners

The following one-liners demonstrate different capabilities:

```
# Files opened by process
bpftrace -e 'tracepoint:syscalls:sys_enter_open { printf("%s %s\n", comm, str(args->filename)); }

# Syscall count by program
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

# Read bytes by process:
bpftrace -e 'tracepoint:syscalls:sys_exit_read /args->ret/ { @[comm] = sum(args->ret); }'

# Read size distribution by process:
bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm] = hist(args->ret); }'

# Show per-second syscall rates:
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @ = count(); } interval:s:1 { print(@); clear(@); }

# Trace disk size by process
bpftrace -e 'tracepoint:block:block_rq_issue { printf("%d %s %d\n", pid, comm, args->bytes); }'

# Count page faults by process
bpftrace -e 'software:faults:1 { @[comm] = count(); }'

# Count LLC cache misses by process name and PID (uses PMCs):
bpftrace -e 'hardware:cache-misses:1000000 { @[comm, pid] = count(); }'

# Profile user-level stacks at 99 Hertz, for PID 189:
bpftrace -e 'profile:hz:99 /pid == 189/ { @[ustack] = count(); }'

# Files opened, for processes in the root cgroup-v2
bpftrace -e 'tracepoint:syscalls:sys_enter_openat /cgroup == cgroupid("/sys/fs/cgroup/unified/mycg")
```

https://github.com/iovisor/bpftrace https://github.com/iovisor/kubectl-trace

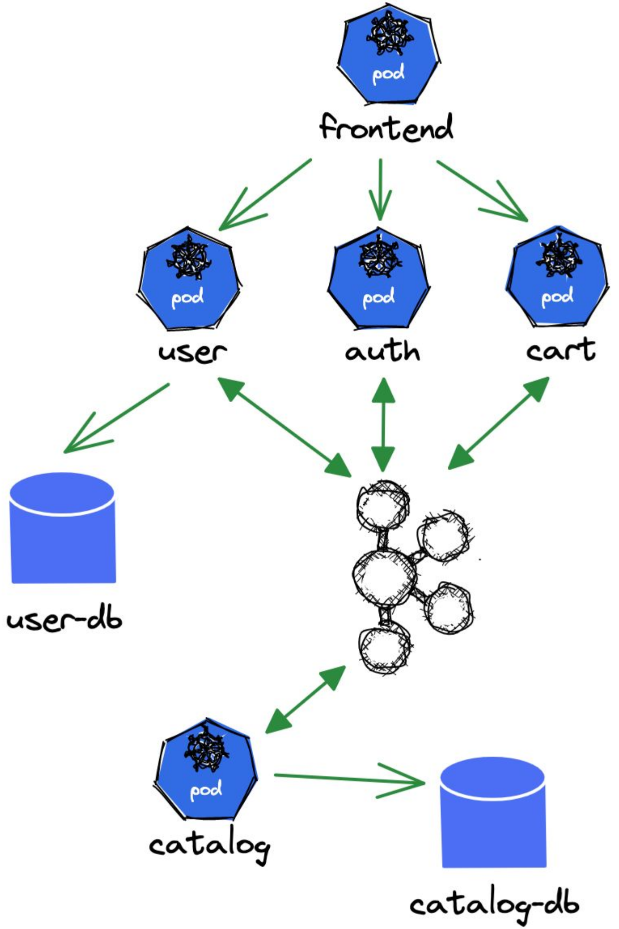
eBPF Exporter



https://github.com/cloudflare/ebpf_exporter

What's Pixie?

- **Open-source CNCF observability platform**
  - Using eBPF
- **Automatically traces network messages**
  - Kafka, HTTP, MySQL, etc.
  - Always active
- **No instrumentation**
  - No code modifications
  - No redeployments

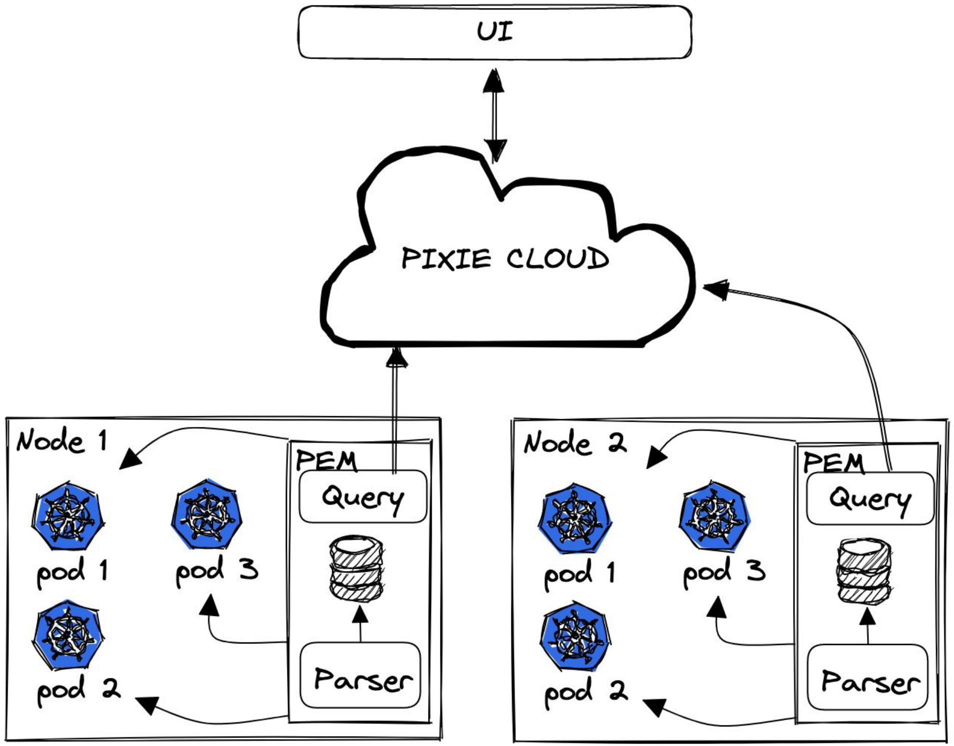


A CNCF sandbox project

CLOUD NATIVE
COMPUTING FOUNDATION

Pixie's approach

- Pixie Edge Module (PEM) deploys on every node
- Capture data with eBPF
- Process data in user-space (protocol parsing)
- Store data into tables for querying by user

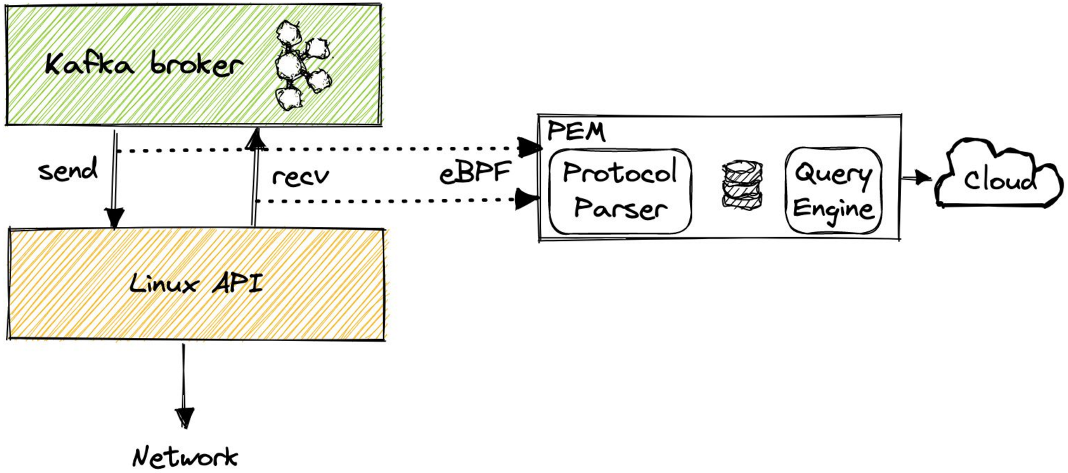


How is the data traced?

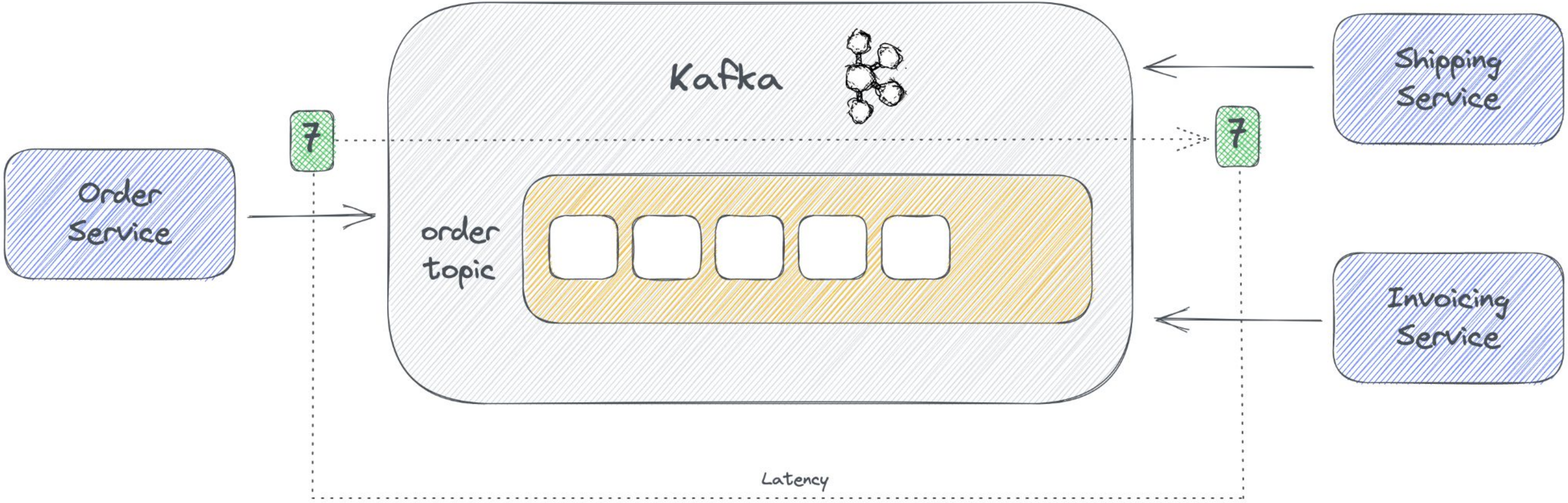
Pixie traces network-related Linux syscalls with eBPF kprobes.

In the case of Kafka:

- Fetch and Produce messages
- JoinGroup and SyncGroup messages etc.



Demo time!



```
bash -c "$(curl -fsSL https://withpixie.ai/install.sh)"
```

Summary

- Kafka observability is challenging
- eBPF opens a new world of possibilities
- Pixie provides auto-instrumentation for Kubernetes applications
  - No code modification
  - No redeployment
  - Easy to use
  - Specific domain metrics
  - Low overhead

Thank you!

Questions?