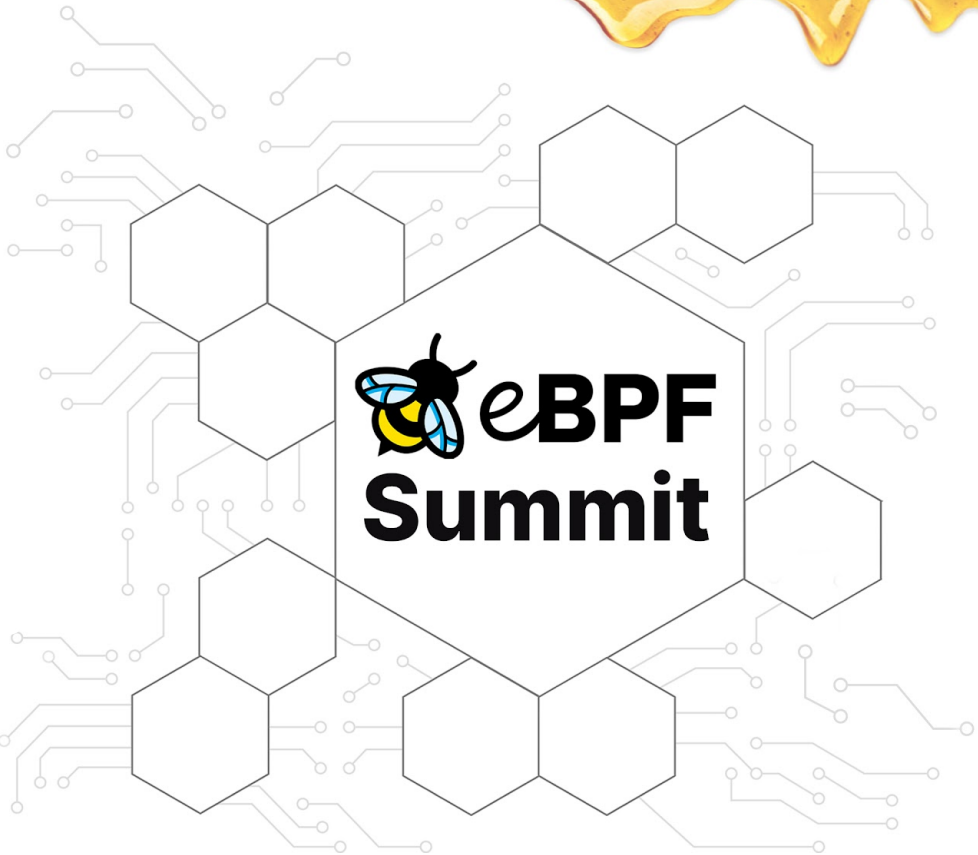
Live Programming and Visualizing eBPF



Nikita Baksalyar

@nbaksalyar

Another way to think about eBPF

- *libbpf* and *bpftrace* are hard to learn
- Command-line tools leave a lot of power unused
  - No interactivity
  - No graphics
  - Limited visualisation

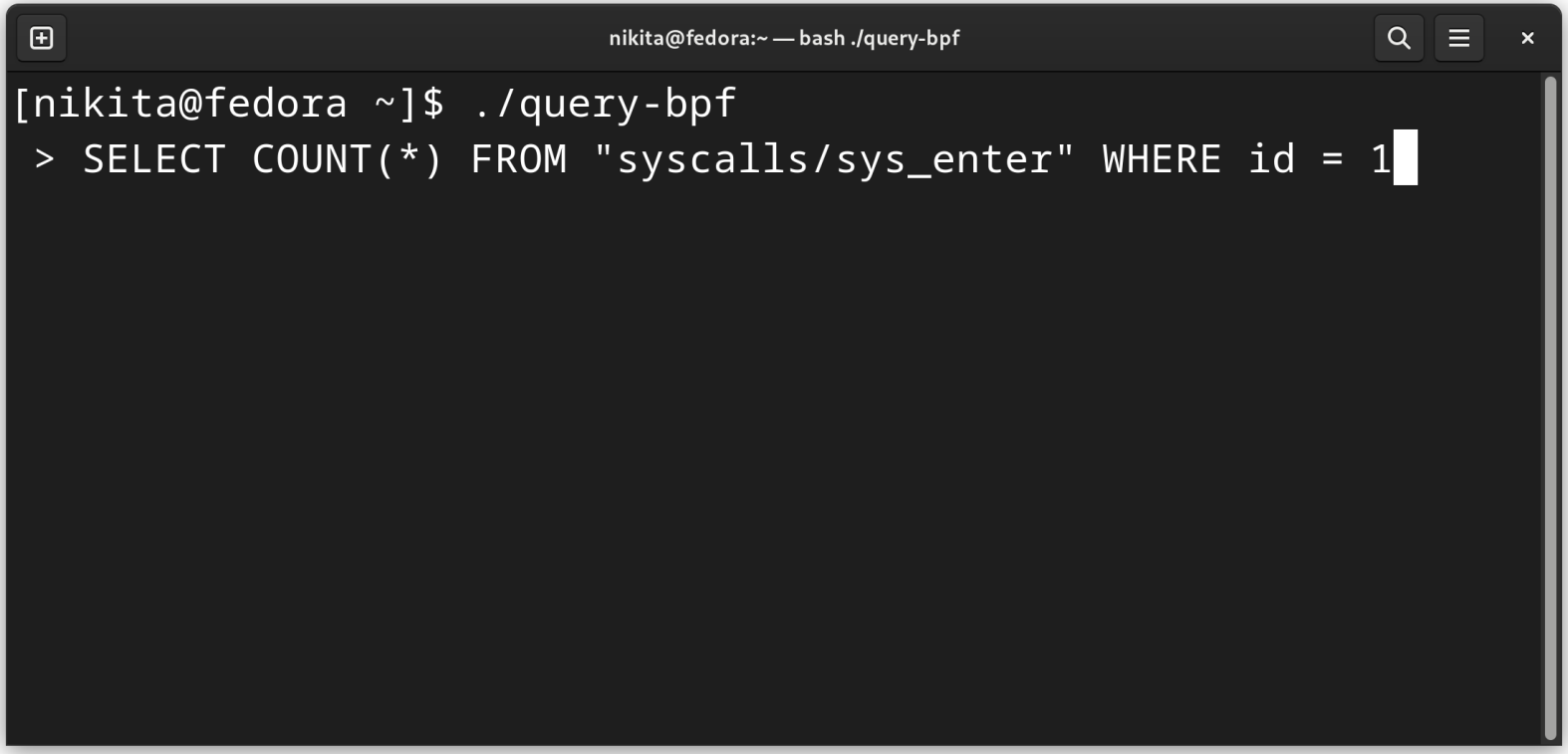
Another way to think about eBPF

- SQL as a source of inspiration
  - Domain-specific language for a database system
  - Code describes *what* you want, not *how* to get it
- Linux kernel as a database?

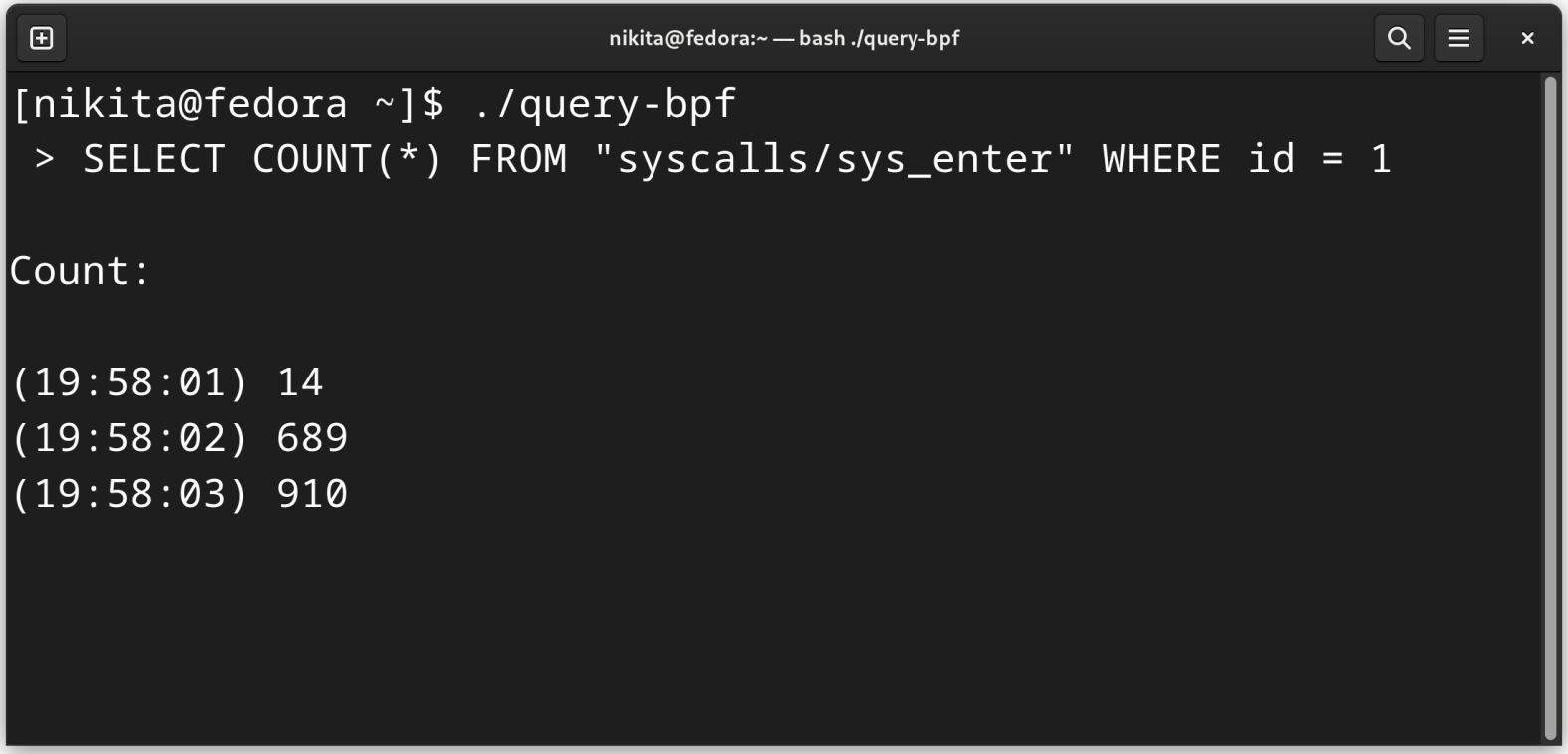
Linux kernel as a database

- Data-centric systems: Apache Spark/Flink
  - Thinking in terms of *streams* of data, not static data
  - SQL-like language can still be used for analytics
- Linux events are also infinite data streams
- Queries can be compiled into eBPF

SQL-like interface for eBPF

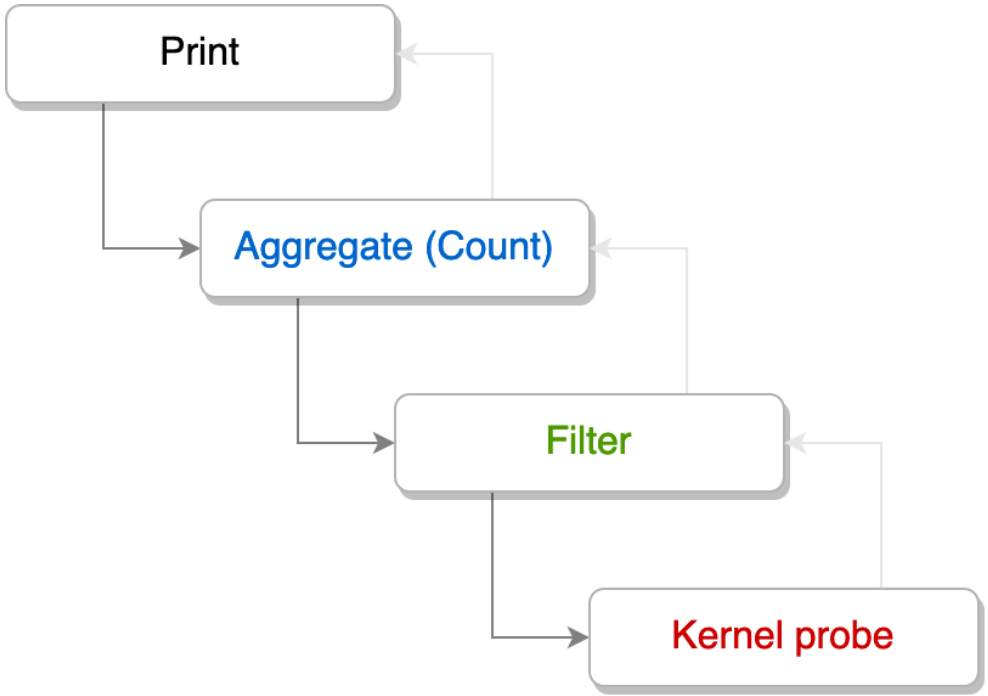


SQL-like interface for eBPF



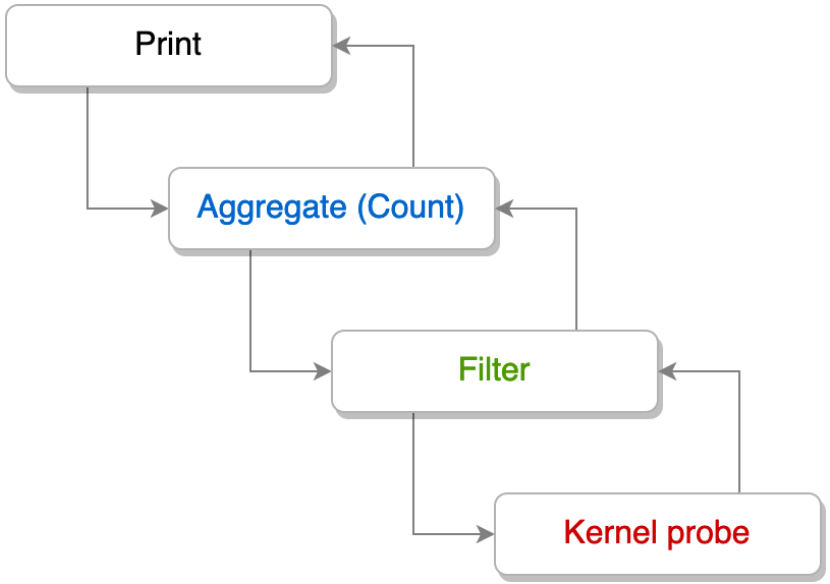
SQL-like interface for eBPF

```
SELECT COUNT(*) FROM “syscalls/sys_enter” WHERE id = 1
```



SQL-like interface for eBPF

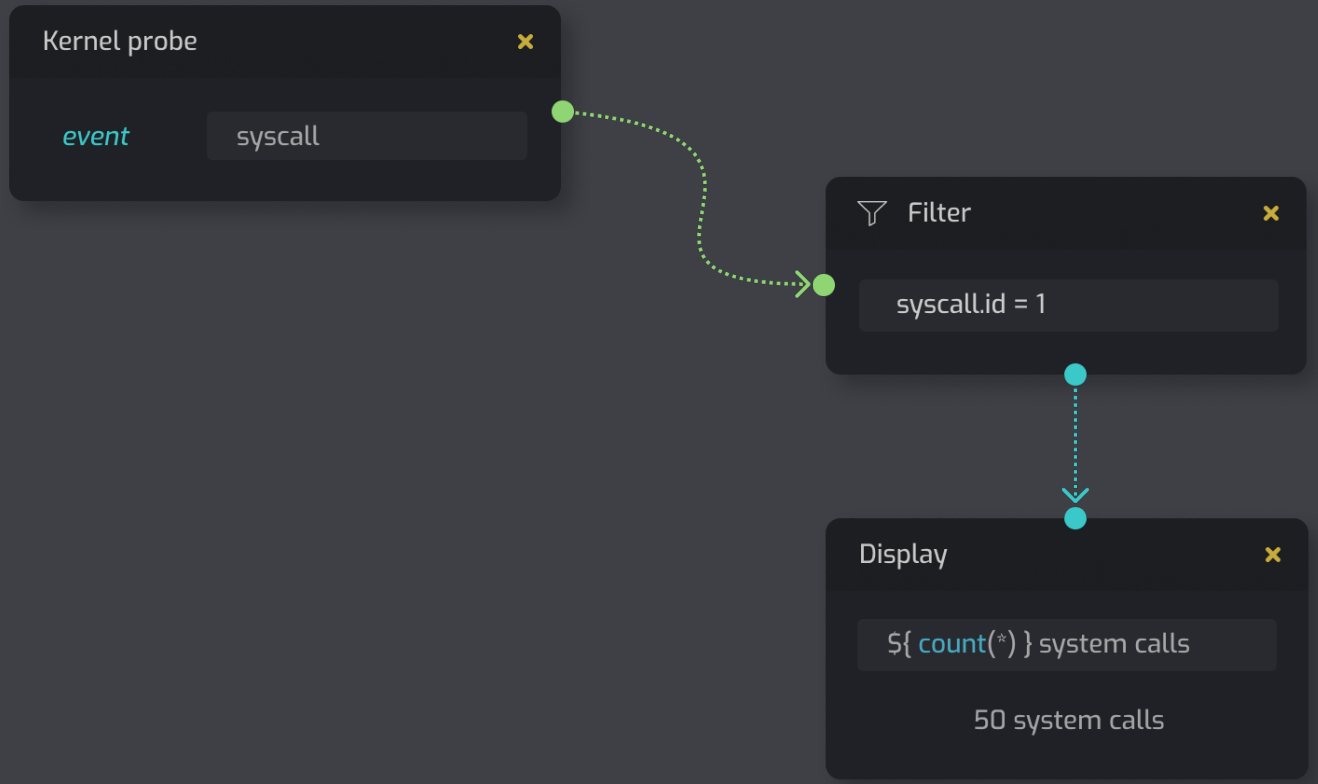
```
SELECT COUNT(*) FROM “syscalls/sys_enter” WHERE id = 1
```



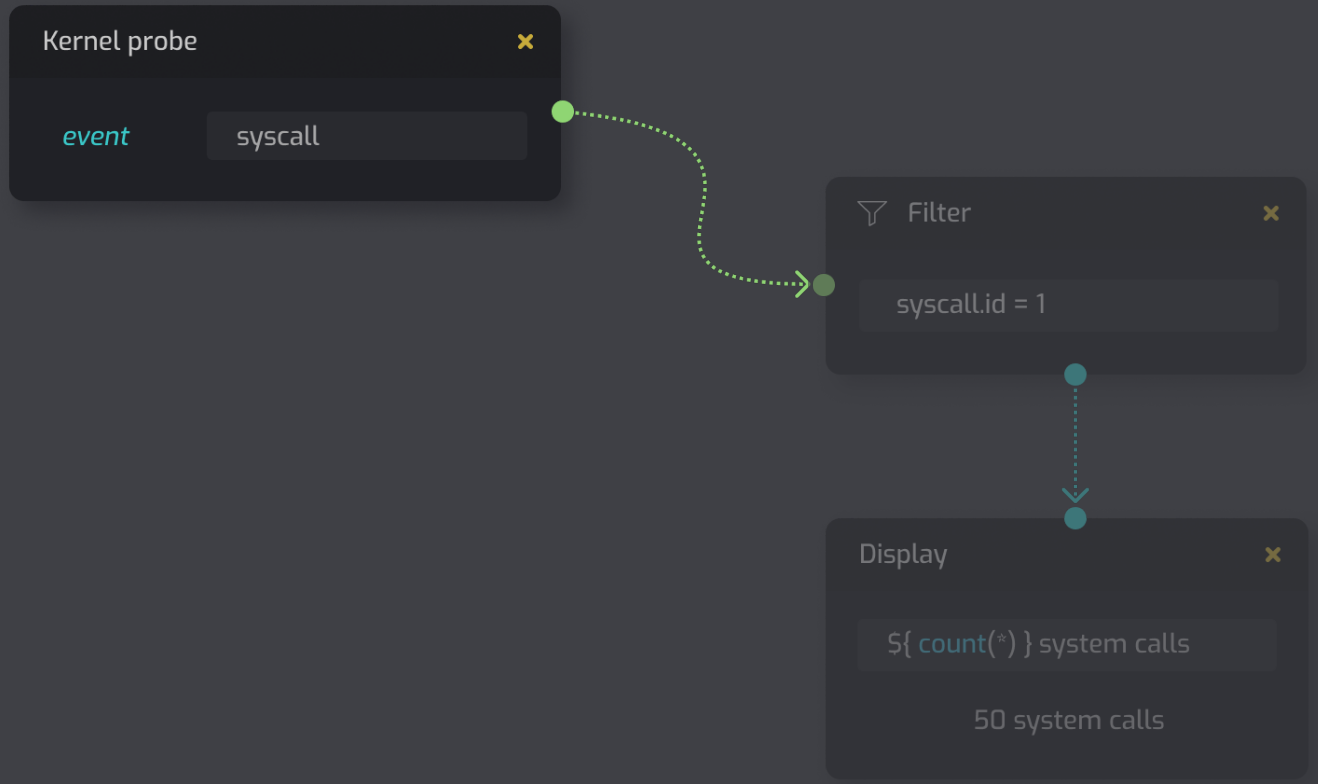

```
@bpf_map
events_count = 0

@kernel_probe(“syscall”)
def event_handler(arg):
  if (arg == 1):
    events_count += 1
```

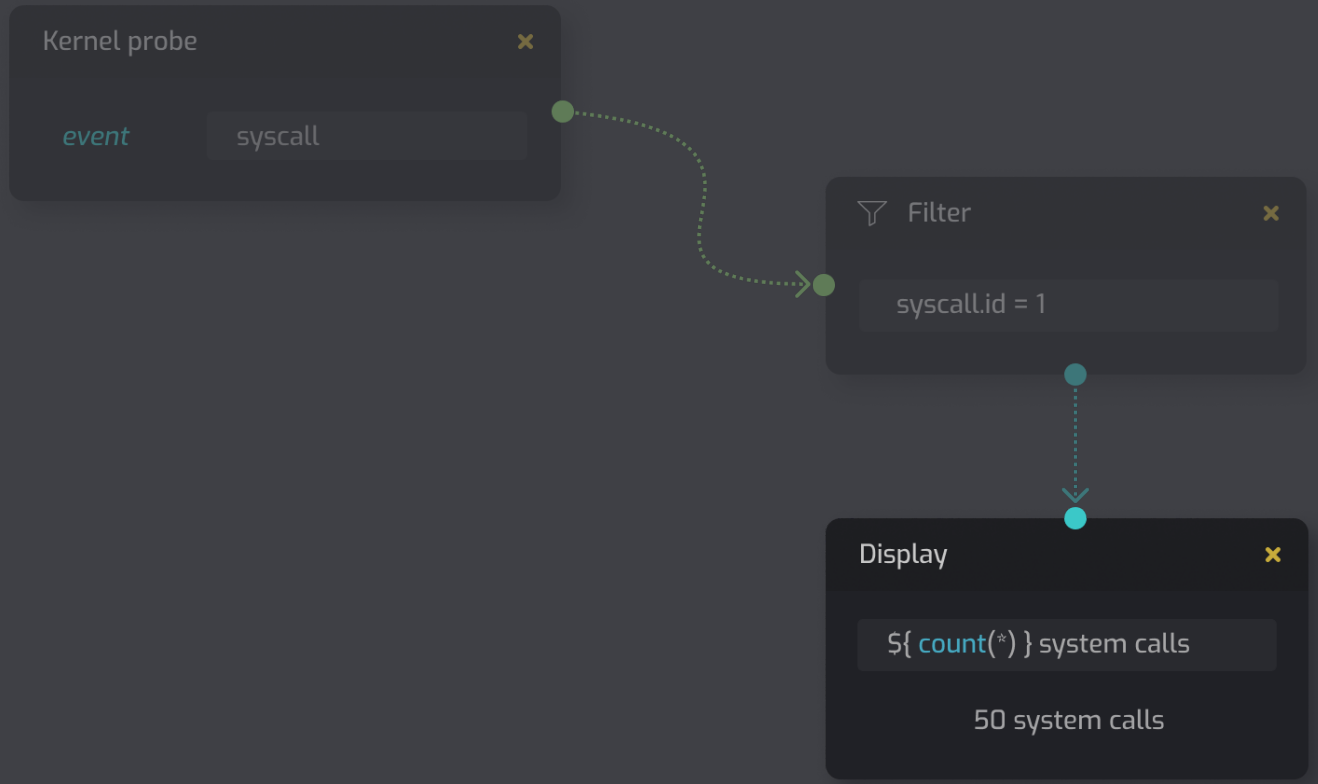
```
SELECT COUNT(*) FROM “syscalls/sys_enter” WHERE id = 1
```



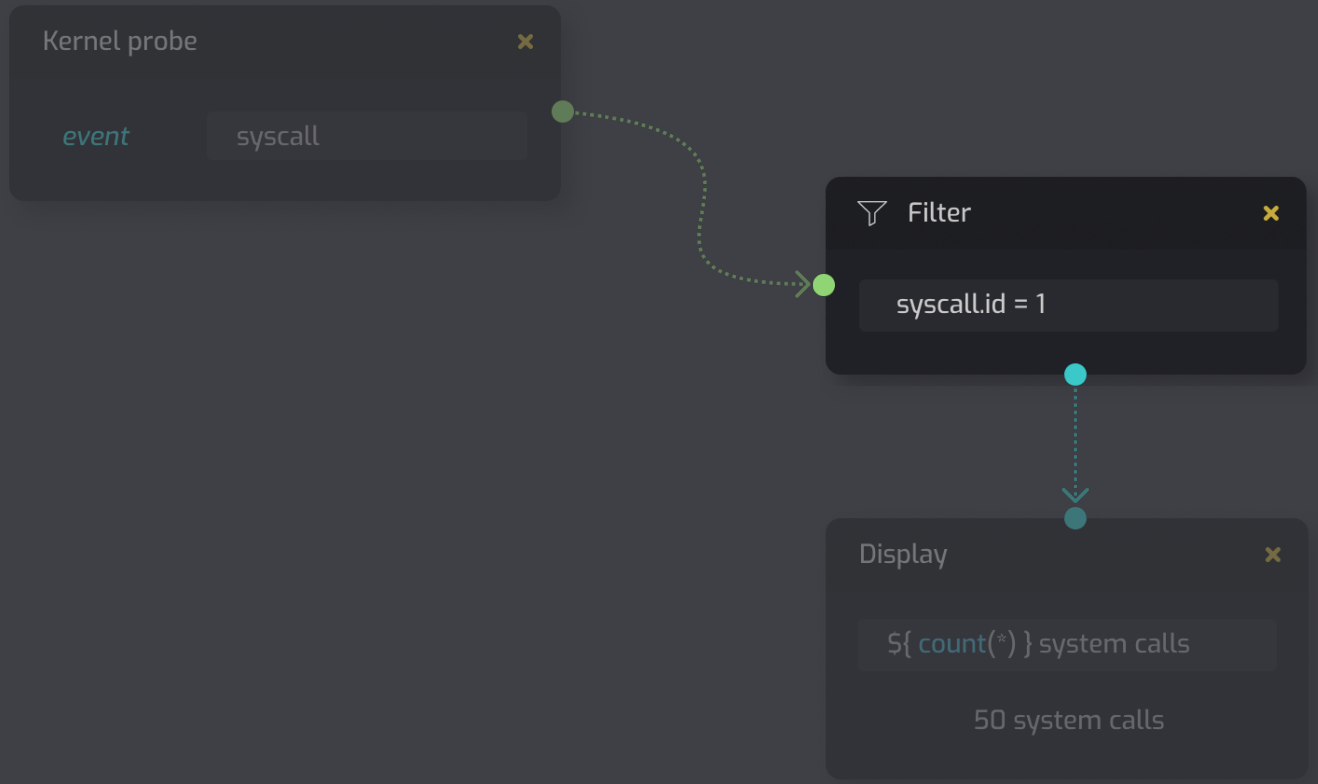
```
SELECT COUNT(*) FROM “syscalls/sys_enter” WHERE id = 1
```

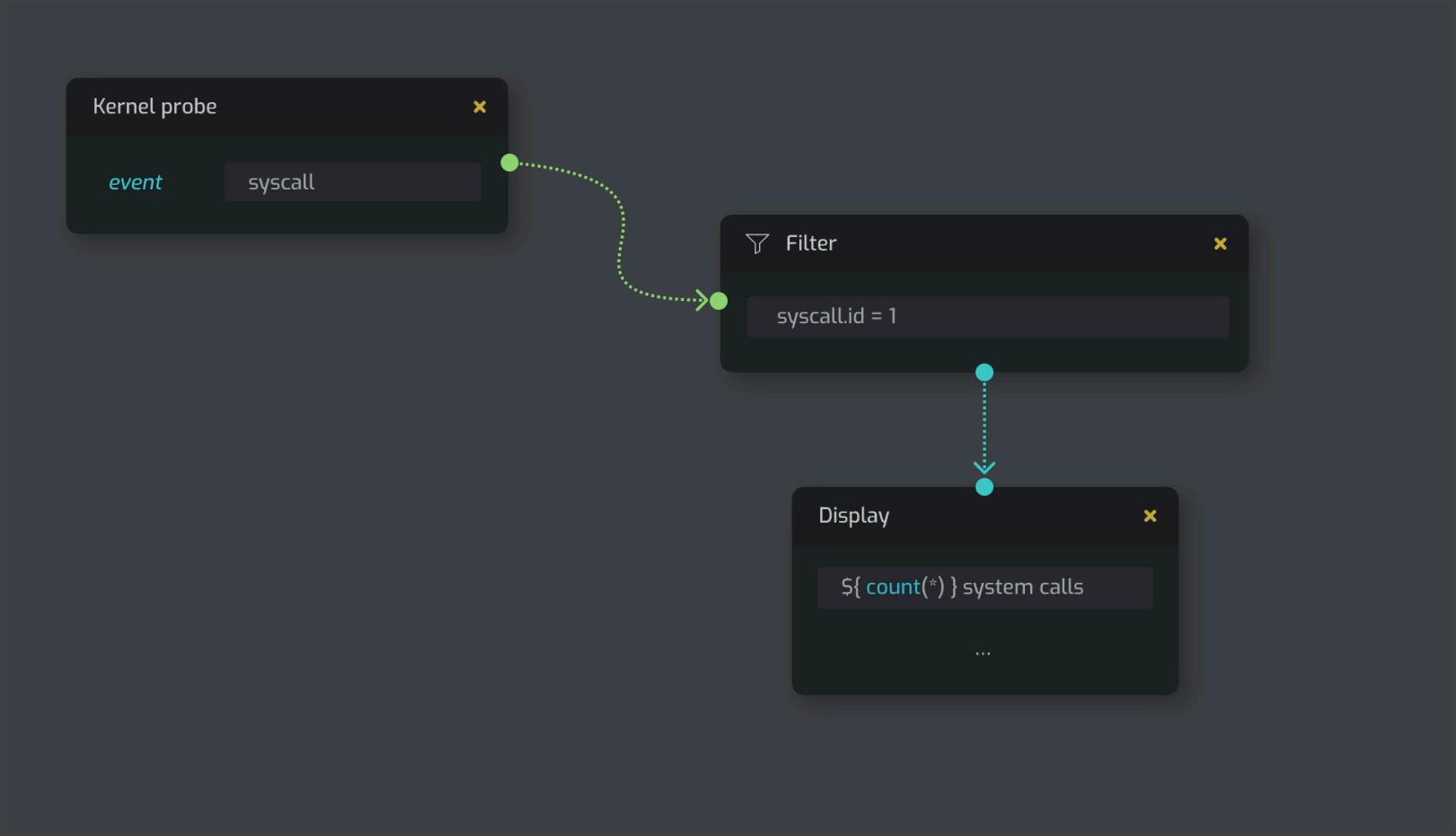


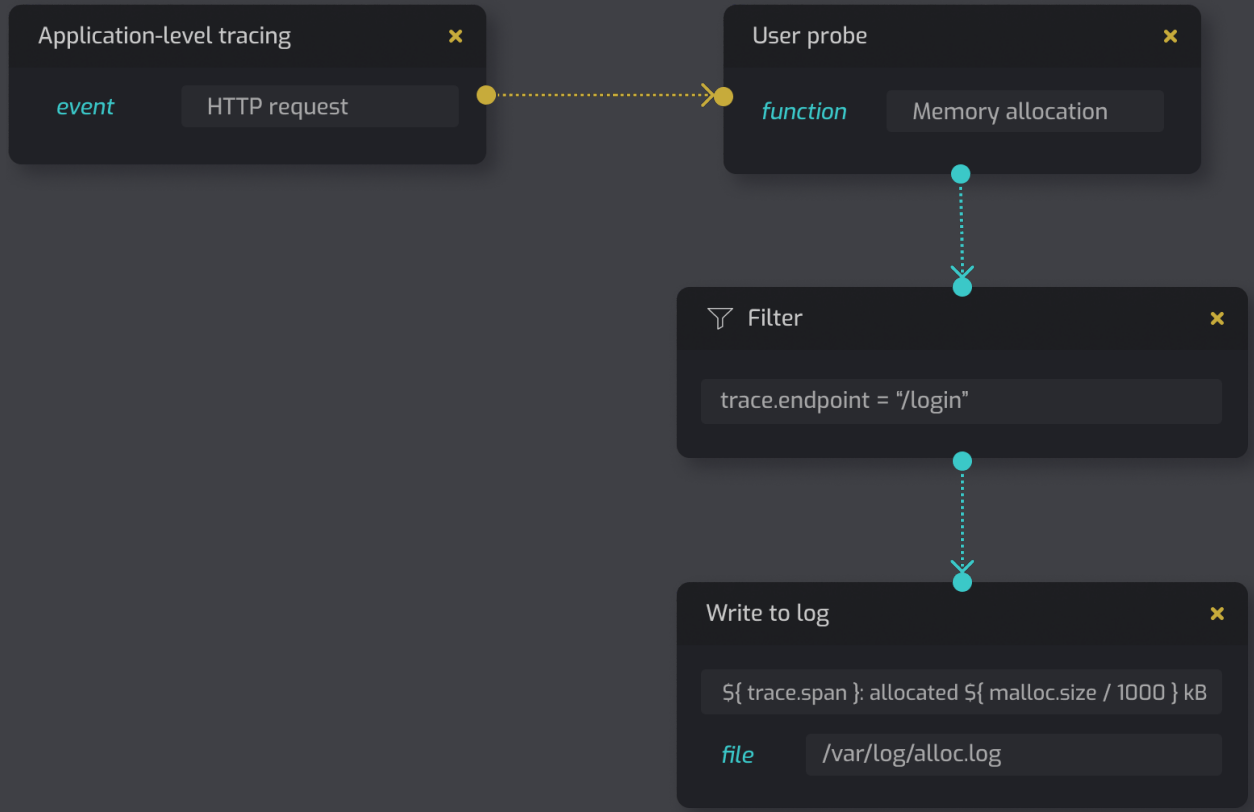
```
SELECT COUNT(*) FROM “syscalls/sys_enter” WHERE id = 1
```

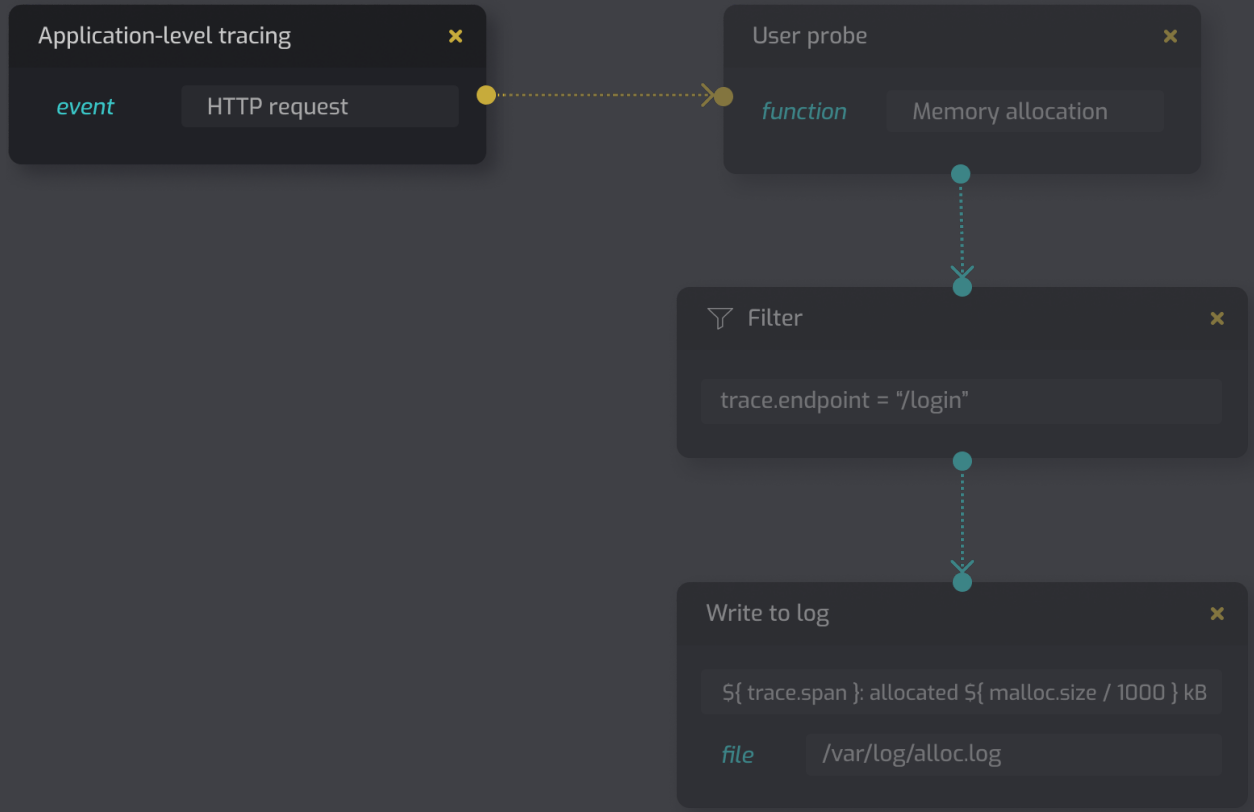


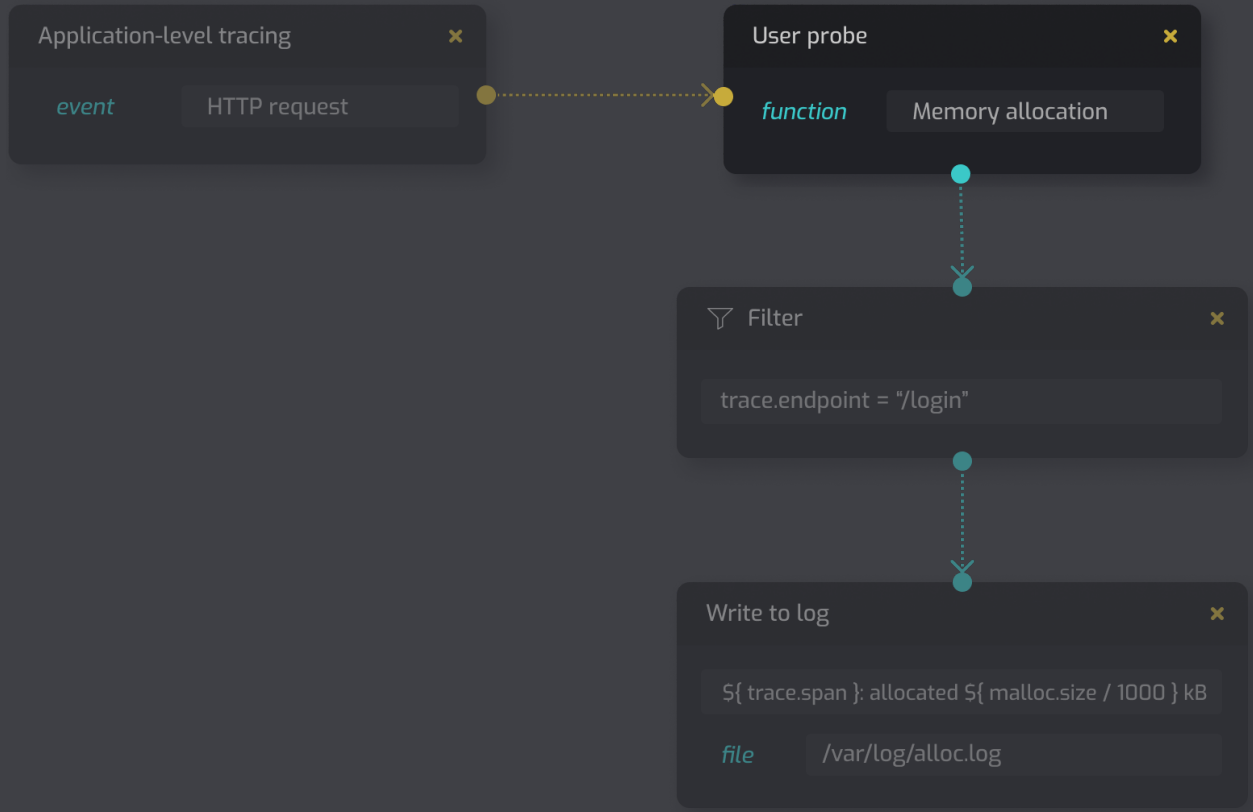
```
SELECT COUNT(*) FROM “syscalls/sys_enter” WHERE id = 1
```

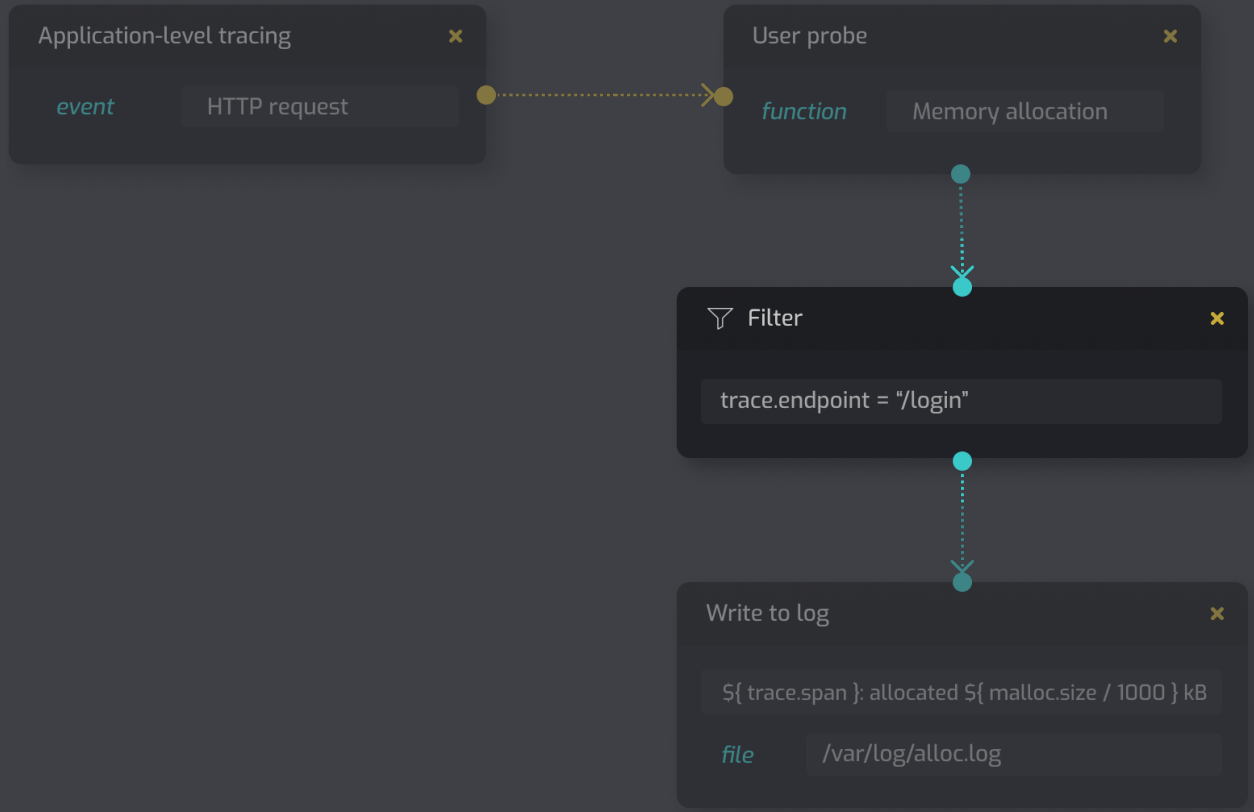


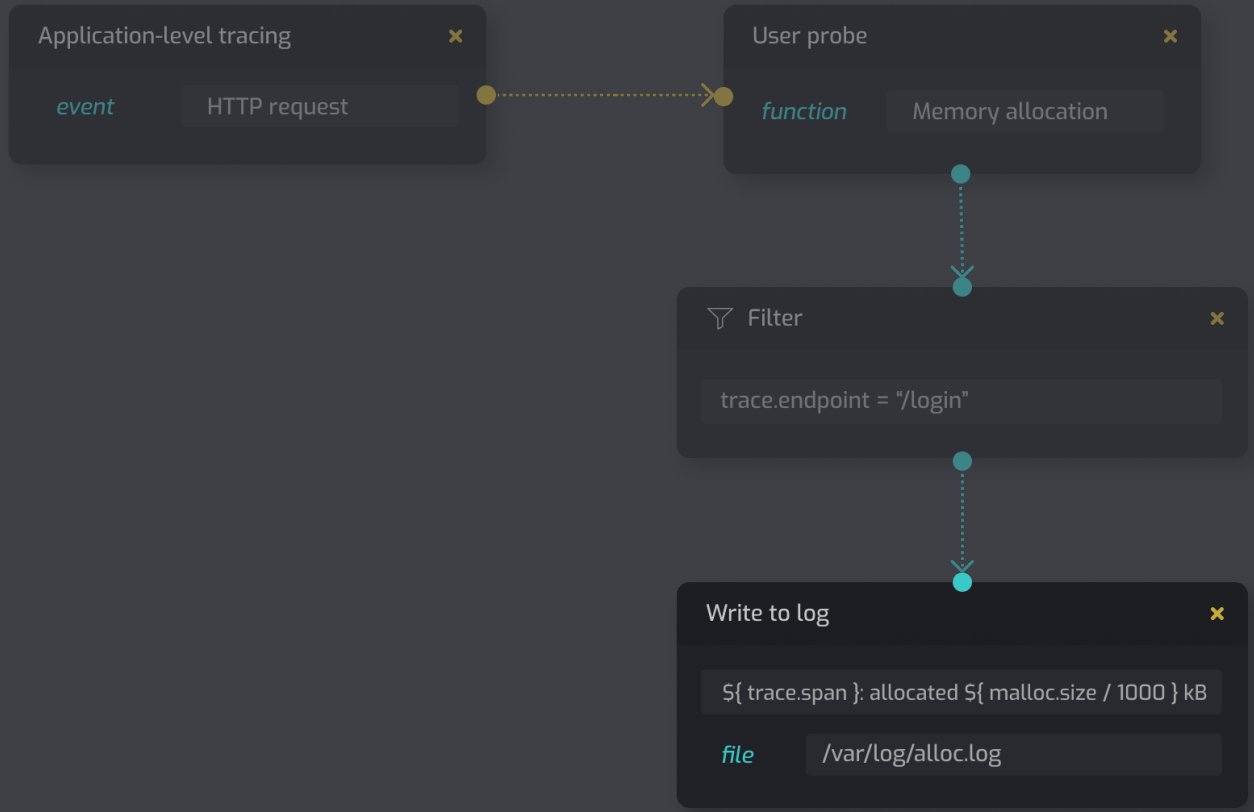

Kernel probe
event
syscall
Filter
syscall.id = 1
Display
${ count(*) } system calls
...

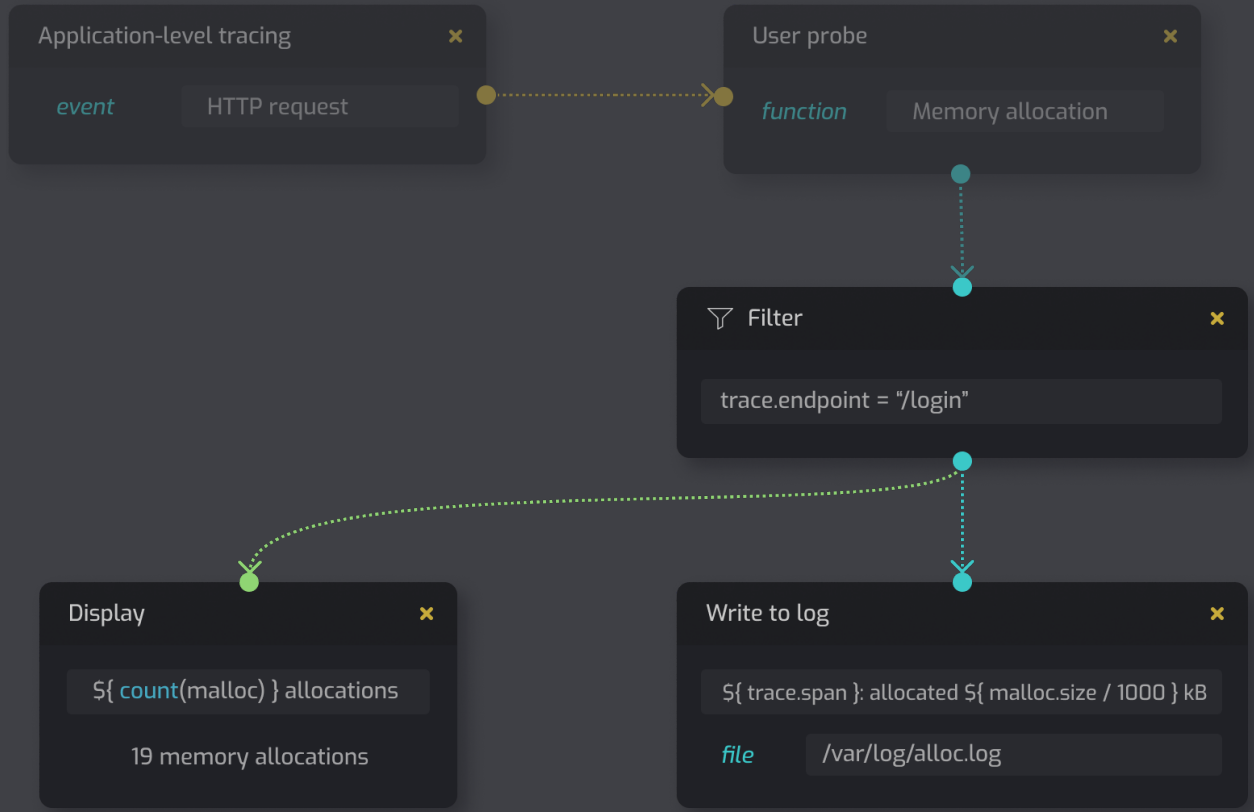

Application-level tracing
event
HTTP request
User probe
function
Memory allocation
Filter
trace.endpoint = "/login"
Write to log
${ trace.span }: allocated ${ malloc.size / 1000 } kB
file
/var/log/alloc.log

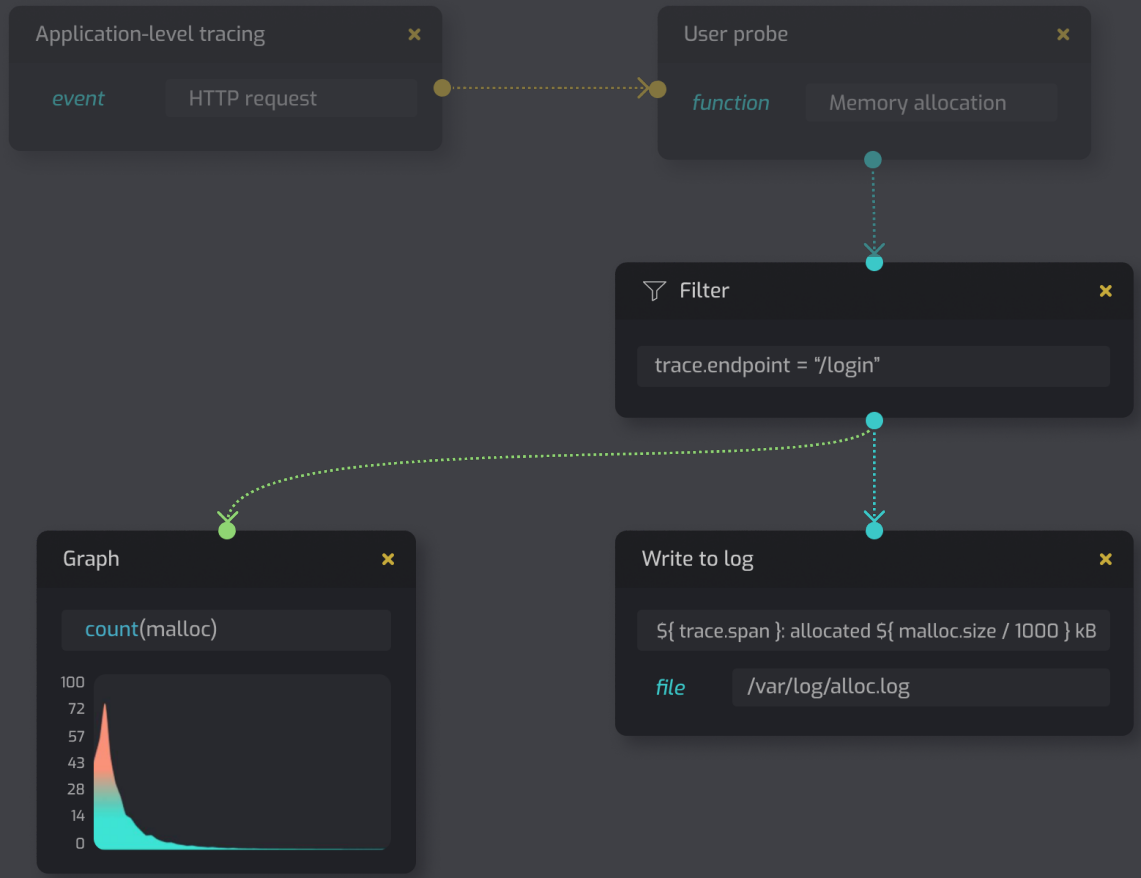

Application-level tracing
event
HTTP request
User probe
function
Memory allocation
Filter
trace.endpoint = "/login"
Write to log
${ trace.span }: allocated ${ malloc.size / 1000 } kB
file
/var/log/alloc.log


Application-level tracing
event
HTTP request
User probe
function
Memory allocation
Filter
trace.endpoint = "/login"
Write to log
${ trace.span }: allocated ${ malloc.size / 1000 } kB
file
/var/log/alloc.log


Application-level tracing
event
HTTP request
User probe
function
Memory allocation
Filter
trace.endpoint = "/login"
Write to log
${ trace.span }: allocated ${ malloc.size / 1000 } kB
file
/var/log/alloc.log


Application-level tracing
event
HTTP request
User probe
function
Memory allocation
Filter
trace.endpoint = "/login"
Write to log
${ trace.span }: allocated ${ malloc.size / 1000 } kB
file
/var/log/alloc.log


Application-level tracing
event
HTTP request
User probe
function
Memory allocation
Filter
trace.endpoint = "/login"
Display
${ count(malloc) } allocations
19 memory allocations
Write to log
${ trace.span }: allocated ${ malloc.size / 1000 } kB
file
/var/log/alloc.log


Application-level tracing
event
HTTP request
User probe
function
Memory allocation
Filter
trace.endpoint = "/login"
Graph
count(malloc)
100
72
57
43
28
14
0
Write to log
${ trace.span }: allocated ${ malloc.size / 1000 } kB
file
/var/log/alloc.log

How does it work?

- Web user interface (TypeScript + React)
- LLVM-based compiler produces eBPF code
- Data can be transferred efficiently through maps
    - Ring buffers <-> WebSocket
- Can be expanded with more custom operators written in Rust

More advantages

- API to compose with existing tools
- Easier debugging for eBPF programs
- IDE-like capabilities
  - Code completion aided by BTF and CO-RE
  - Snippets and patterns
- Efficiency
  - Optimal data exchange with ad-hoc data structures
  - Database-like algebraic optimizations

Conclusion

- eBPF is hard but we can make it more accessible!
- Visualization can be a powerful tool
- Follow the open source project development:

  https://github.com/nbaksalyar/metalens

Thank you!

Please ask your questions on Slack.