SNIA DEVELOPER CONFERENCE
SDC 24
BY Developers FOR Developers

# Read performance Strategies for Workload using EBPF

Yadavendra Yadav
Software Architect
IBM

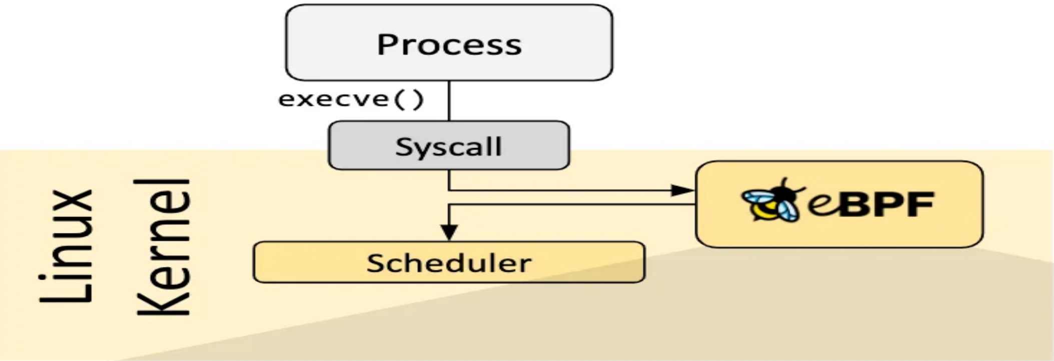
# Agenda:

- Introduction to eBPF
- Architecture of eBPF
- Linux Kernel ReadAhead
- Retrieval Augmented Generation (RAG) for LLMs
- Demo

# Introduction to eBPF

- eBPF is a revolutionary kernel technology that allows developers to write custom code that can be loaded into the kernel dynamically, changing the way the kernel behaves.
- BPF evolved to what we call “extended BPF” or “eBPF” starting in kernel version 3.18

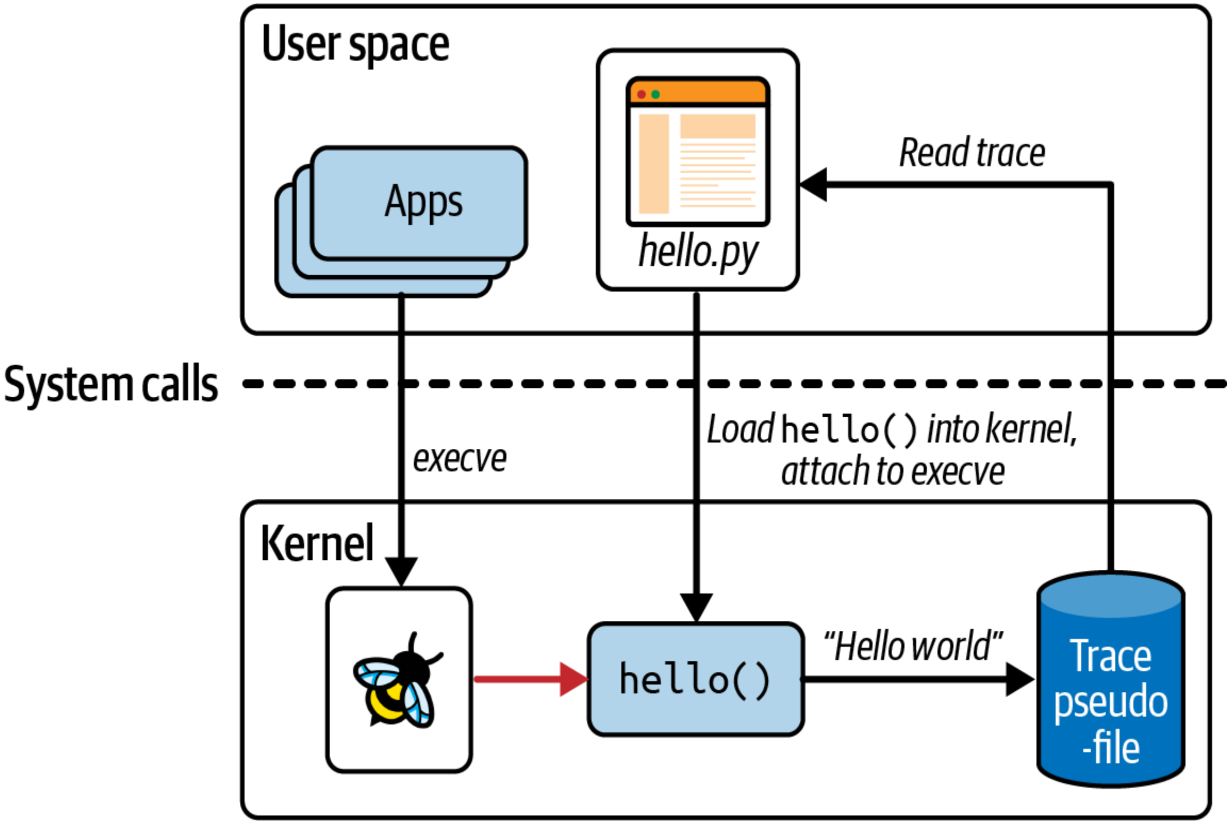
*Tcpdump using BPF*
*ldh [12]*
*jeq #ETHERTYPE IP, L1, L2*
*L1: ret #TRUE*
*L2: ret #0*




```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

# Introduction to eBPF

```
#!/usr/bin/python
from bcc import BPF
program = r"""
int hello(void *ctx) {
    bpf_trace_printk("Hello World!");
    return 0;
}
"""
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall,
fn_name="hello"

b.trace_print()
```



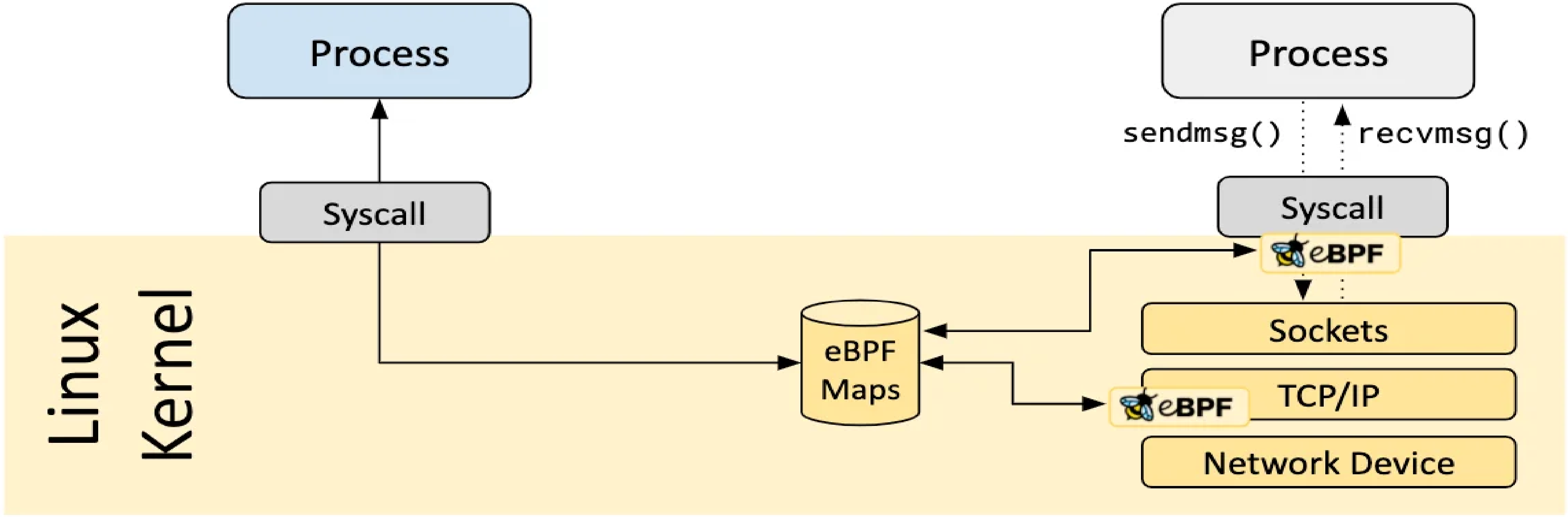

**Kernel Module:**

The biggest challenge here is that this is still full-on kernel programming. Users have historically been very cautious about using kernel modules, for one simple reason: if kernel code crashes, it takes down the machine and everything running on it. How can a user be confident that a kernel module is safe to run?

# Introduction to eBPF

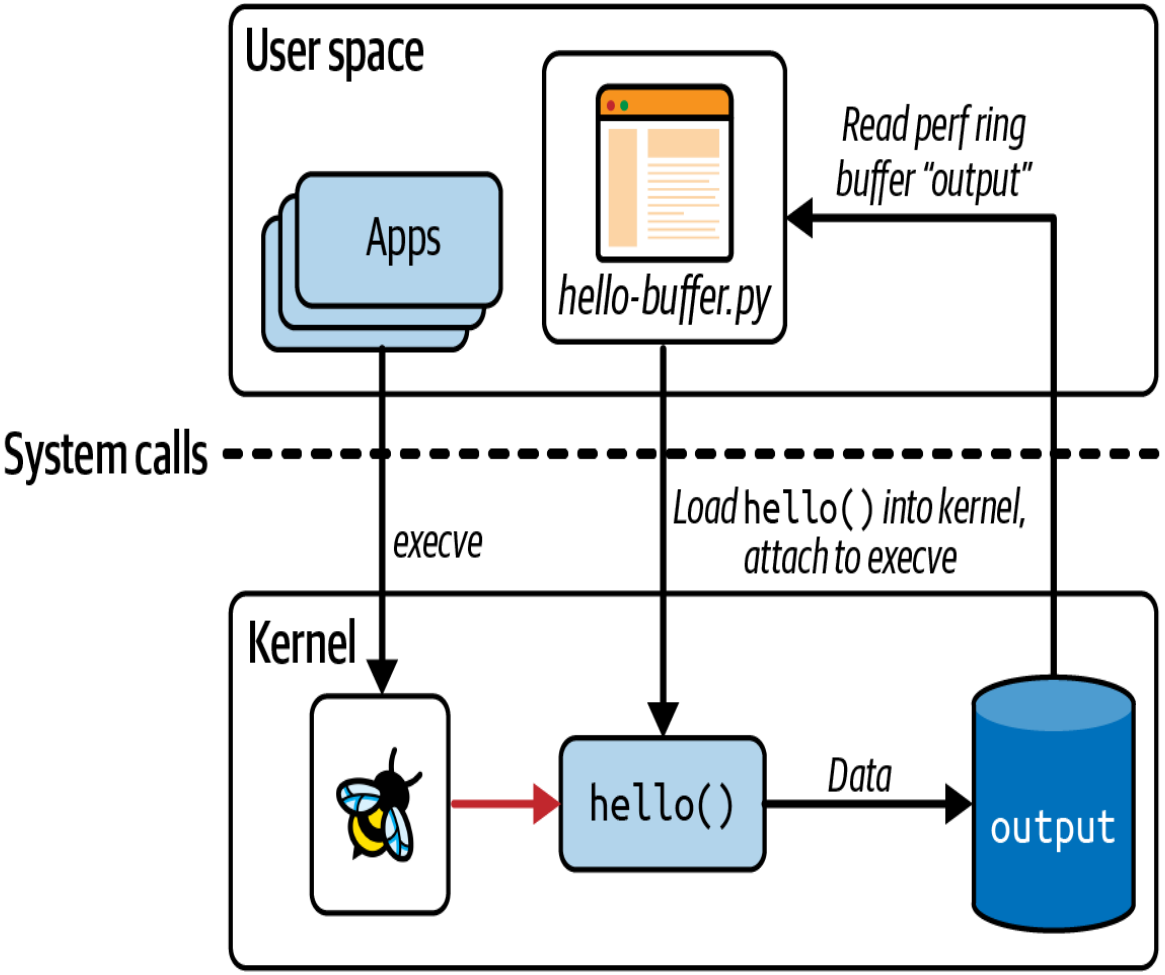
### BPF Maps

- Hash tables, Arrays
- LRU (Least Recently Used)
- Ring Buffer
- Stack Trace
- LPM (Longest Prefix match)



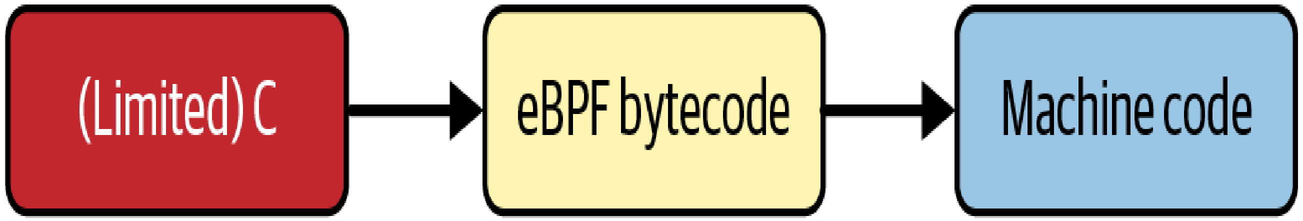
# Introduction to eBPF

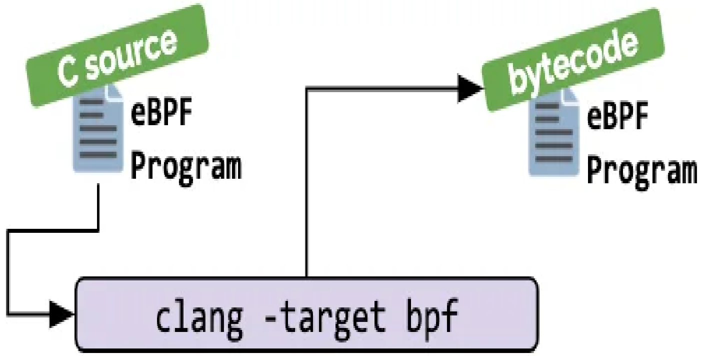
```
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")

def print_event(cpu, data, size):
   data = b["output"].event(data)
   print(f"{data.pid} {data.uid}
{data.command.decode()} " + \
         f"{data.message.decode()}")

b["output"].open_perf_buffer(print_event)
while True:
   b.perf_buffer_poll()
```



# Architecture to eBPF

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>
#include "bootstrap.h"

char LICENSE[] SEC("license") = "Dual BSD/GPL";

struct {
		__uint(type, BPF_MAP_TYPE_HASH);
		__uint(max_entries, 8192);
		__type(key, pid_t);
		__type(value, u64);
} exec_start SEC(".maps");

SEC("tp/sched/sched_process_exec")
int handle_exec(struct trace_event_raw_sched_process_ex
{
		struct task_struct *task;
		unsigned fname_off;

...
...
SEC("tp/sched/sched_process_exit")
int handle_exit(struct trace_event_raw_sched_process_template
*ctx)
{
```



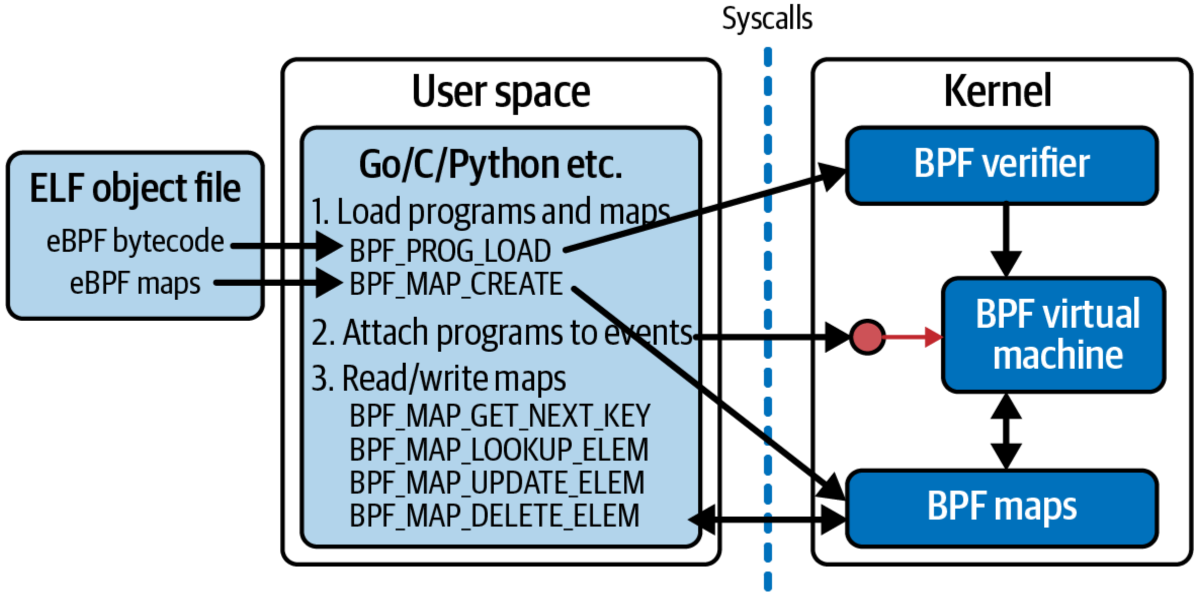
# Architecture to eBPF

**BPF system call:**

int bpf(int cmd, union bpf_attr *attr, unsigned int size);

***bpf(BPF_MAP_CREATE**, {map_type=BPF_MAP_TYPE_HASH, key_size=4, value_size=12, max_entries=10240... map_name="config", ...btf_fd=3,...}, 128) = 5*

***bpf(BPF_PROG_LOAD**, {prog_type=BPF_PROG_TYPE_KPROBE, insn_cnt=44, insns=0xffffa836abe8, license="GPL", ... prog_name="hello", ... expected_attach_type=BPF_CGROUP_INET_INGRESS, prog_btf_fd=3,...}, 128) = 6*



# Architecture to eBPF

## CO-RE, BTF, and Libbpf

**BTF :**

BTF is a format for expressing the layout of data structures and function signatures.
In CO-RE it's used to determine any differences between the structures used at compilation time and at runtime

**Kernel headers:**

eBPF programmers can choose to include individual header files or can use bpftool to generate a header file called *vmlinux.h* from a running system.

**Compiler support :**

clang compiler was enhanced such that, it includes what are known as *CO-RE relocations*, derived from the BTF information describing the kernel data structures.

**Library support for data structure relocations :**

At the point where a user space program loads an eBPF program into the kernel, the CO-RE approach requires the bytecode to be adjusted to compensate for any differences between the data structures present when it was compiled, and what's on the destination machine where it's about to run, based on the CO-RE relocation information compiled into the object.

**Optionally, a BPF skeleton:**

A skeleton can be auto-generated from a compiled BPF object file, containing handy functions that user space code can call to manage the lifecycle of eBPF program.

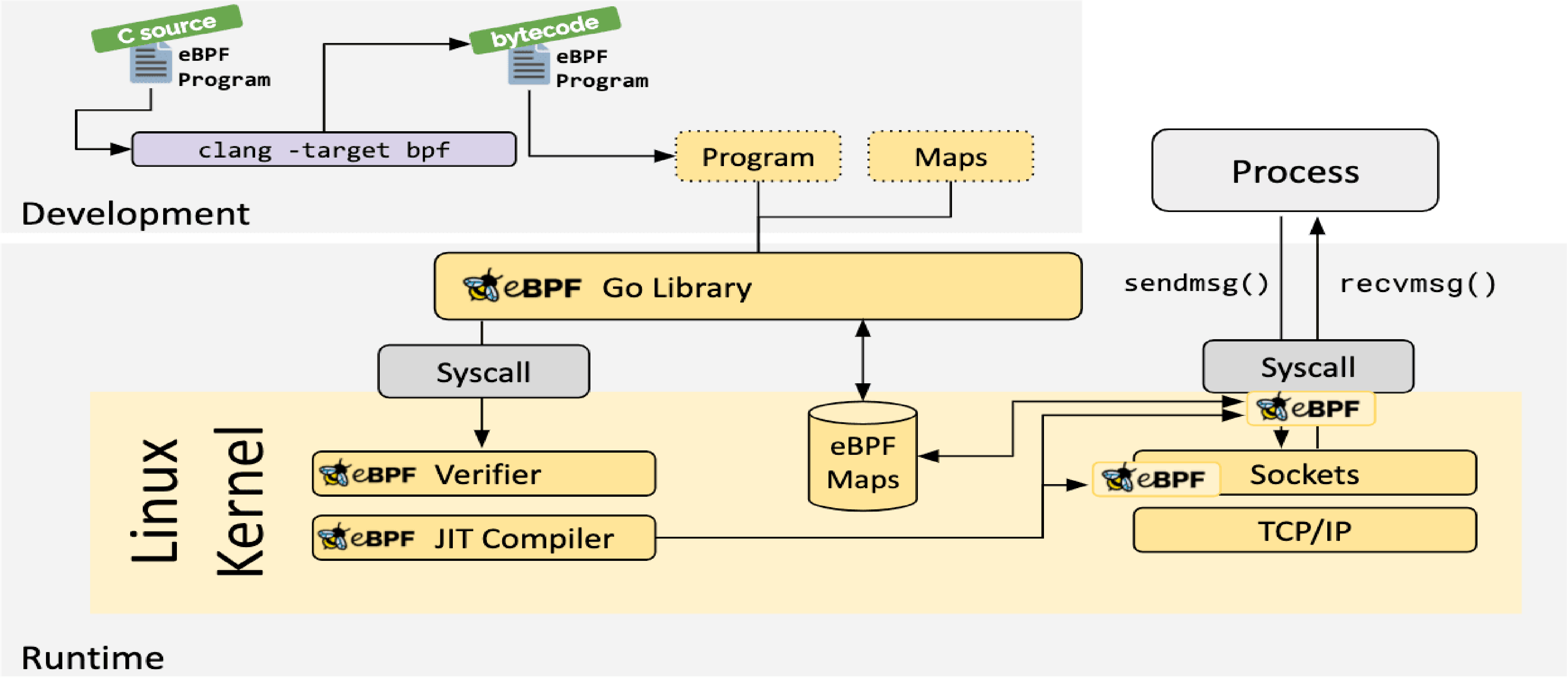
# Architecture to eBPF

## CO-RE, BTF, and Libbpf

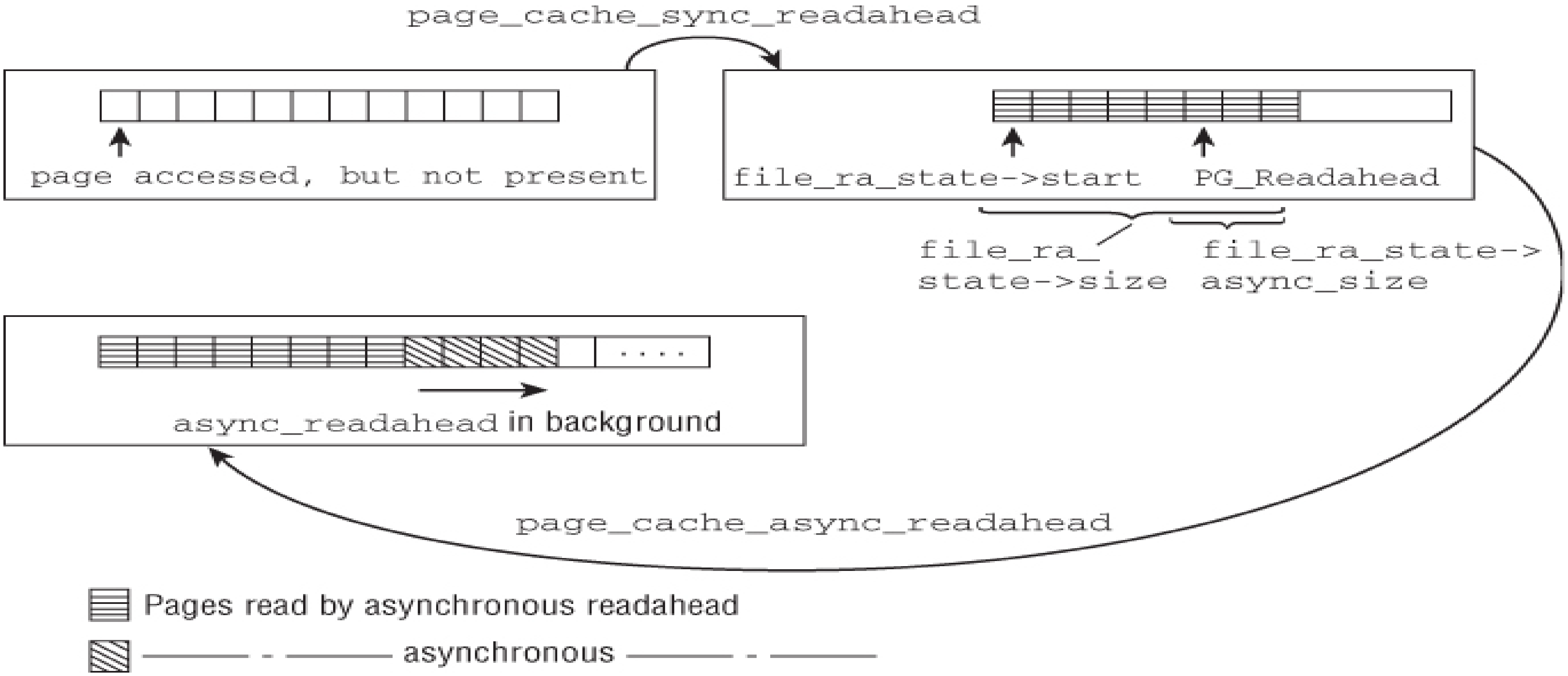
```
SEC("ksyscall/execve")
int BPF_KPROBE_SYSCALL(hello, const char *pathname)
{
…
…
  p = bpf_map_lookup_elem(&my_config, &data.uid);
  if (p != 0) {
     bpf_probe_read_kernel(&data.message, sizeof(data.message), p->message);
  } else {
     bpf_probe_read_kernel(&data.message, sizeof(data.message), message);
  ….
```

```
#define bpf_core_read(dst, sz, src) \ bpf_probe_read_kernel(dst, sz, \
(const void *)__builtin_preserve_access_index(src))
```

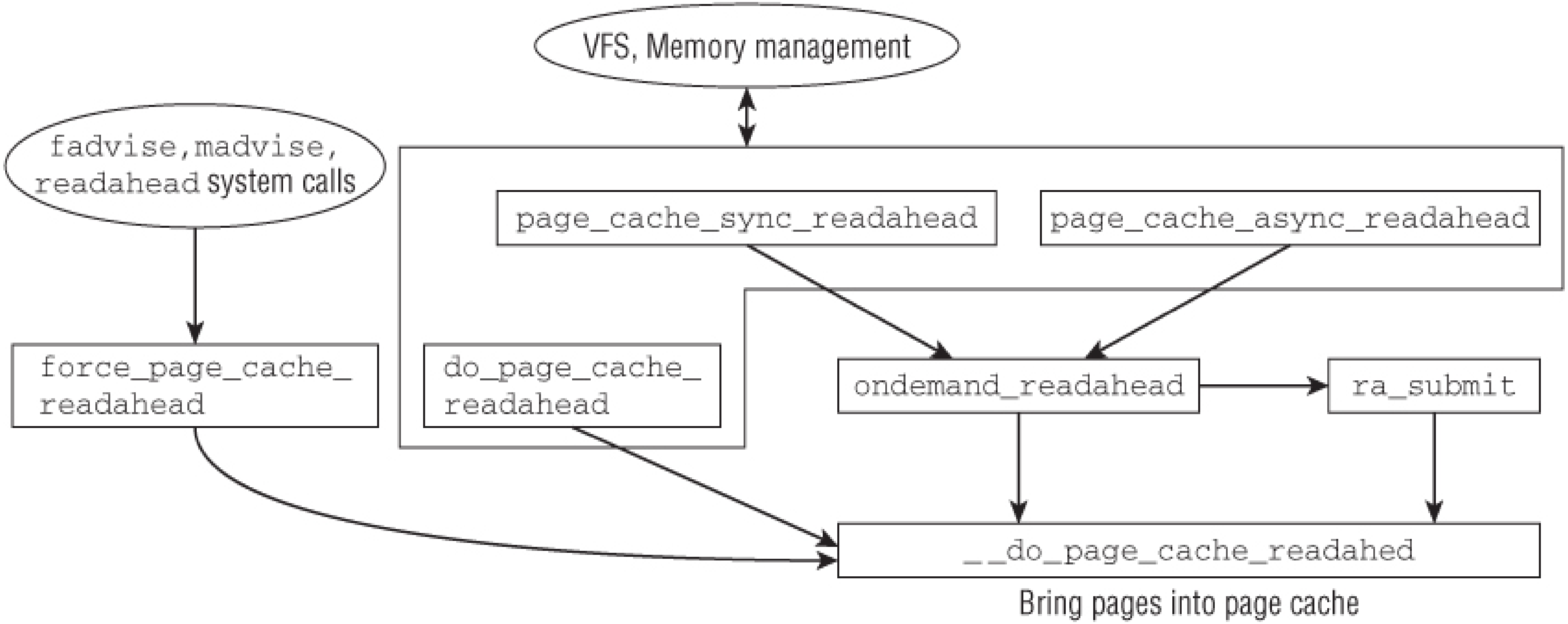
# Architecture to eBPF



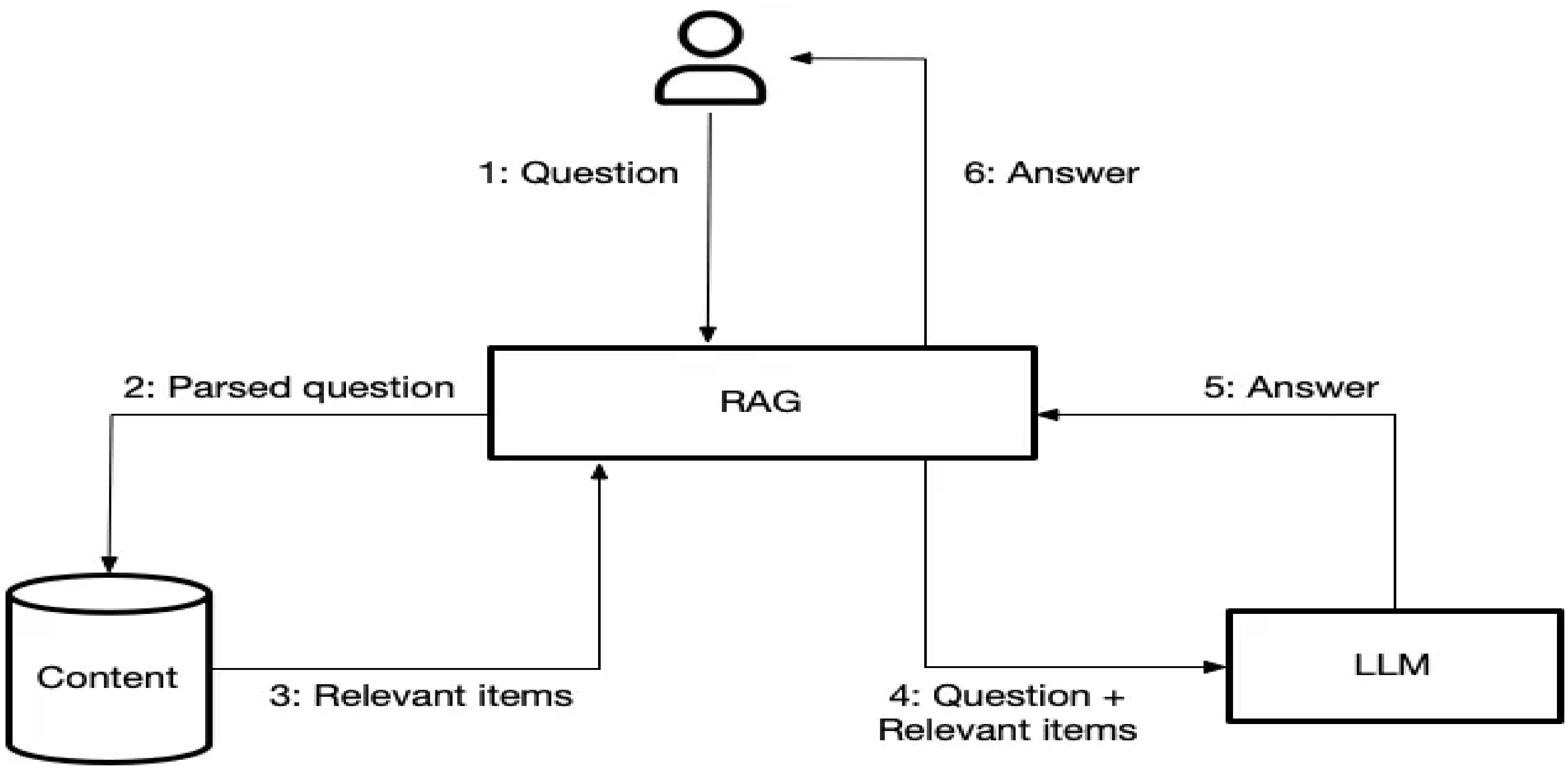
# Linux Kernel ReadAhead



# Linux Kernel ReadAhead



# LLM & RAG (retrieval augmented generation



**Demo:**