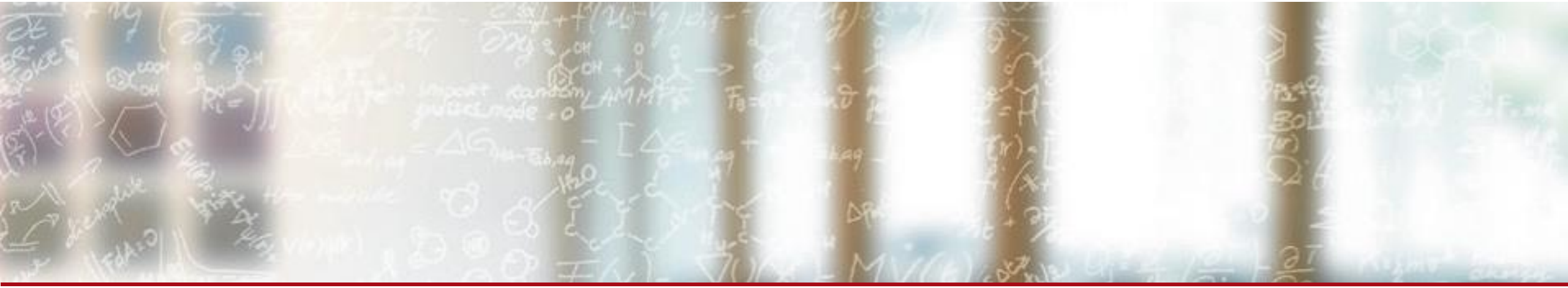
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



A glimpse of YAULT

Victor Holanda Rusu and Massimo Benini, CSCS

May 5th, 2025

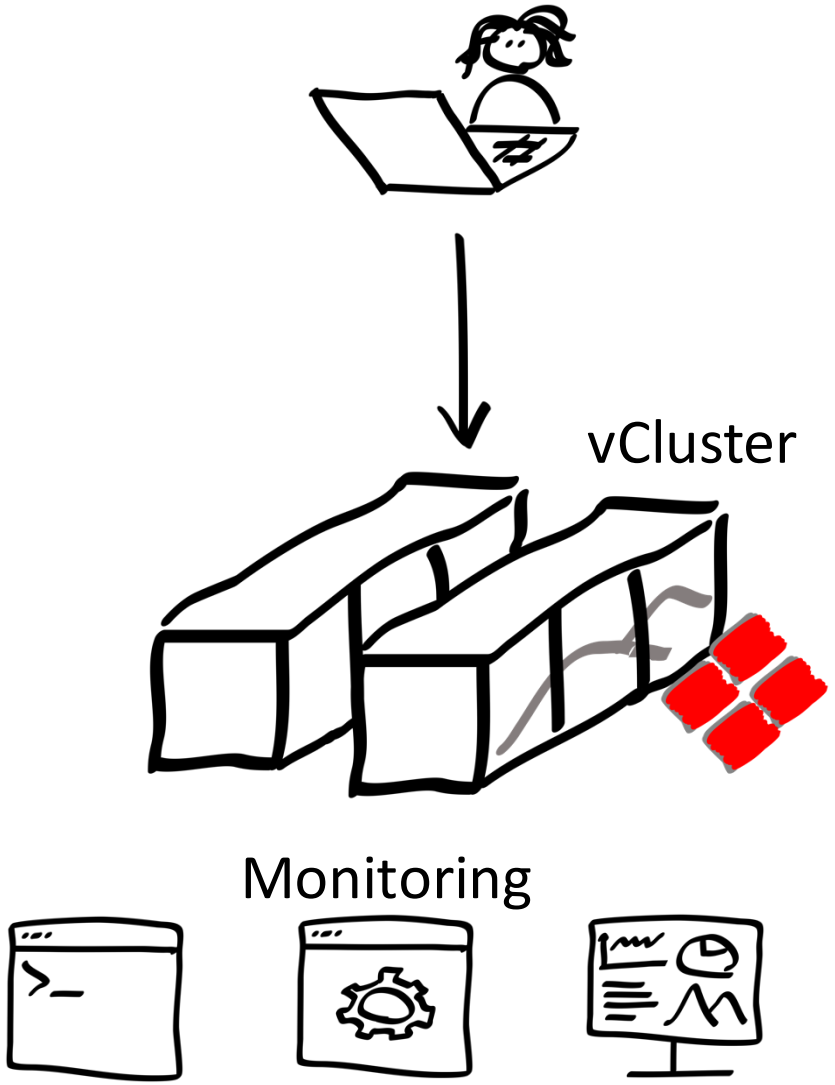
Computing Horizons: CUG 2025

Beyond a System Monitoring

50’000 Feet View Above Our Monitoring

- Infrastructure monitoring
  - Networking
  - Storage
  - Nodes
- Slurm job
  - Jobcomp
  - Slurm DB
- Specific user commands execution
  - auditd
  - syscalls
  - sudo

Can we do more?



Beyond a System Monitoring

50’000 Feet View Above User Application Monitoring

- Kernel namespace or username monitoring
- Usernamespace monitoring generally uses ld preload techniques
  - ELF tricks
  - Able to execute programs as the user

- Kernel namespace monitoring
  - Daemons reading information provided by the kernel
  - Custom kernel modules
  - Application or Kernel tracing
  - EBPF (since 2015)

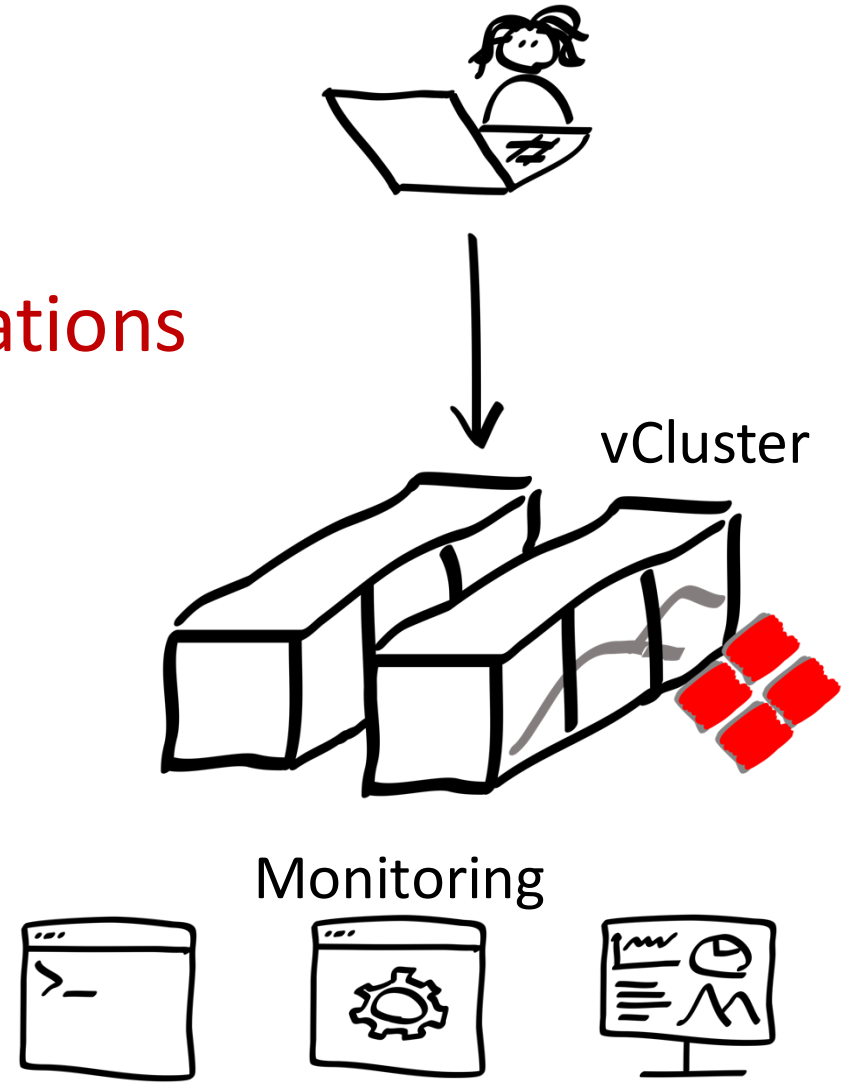
- Limited by the environment
  - Mandatory Access control
  - Chrooted (container)
  - User owns their own environment
- Limited by the Kernel context

Beyond a System Monitoring

50’000 Feet View Above Our Monitoring

- Infrastructure monitoring
  - Networking
  - Storage
  - Nodes
- Slurm job
  - Jobcomp
  - Slurm DB
- Specific User commands execution
  - auditd
  - syscalls
  - sudo
  - eBPF

Can we identify the applications used by the users?



What's eBPF?

TL;DR Edition

- It is a technology that allows programs to run code within a virtual machine inside the Linux kernel
- No need to add additional modules or modify the kernel source code
- Sandboxed environment
- Limited C-based DSL
  - No memory allocation
  - No infinite loops
  - No sleeps
- Library interface for several languages, e.g. Python, Go, C/C++, etc.
- Write modules using a AWK-like language (bpftrace)

What's eBPF?

How does the code look like?

```
#ifndef BPFTRACE_HAVE_BTF
#include <linux/sched.h>
#endif

BEGIN
{
    printf("%-15s %-7s %-7s %s\n", "TIME", "PID", "PPID", "ARGS");
}

tracepoint:syscalls:sys_enter_exec*
{
    $task = (struct task_struct *)curtask;
    printf("%15s %-7d %-7d ", strftime("%H:%M:%S.%f", nsecs), pid, $task->real_parent->pid);
    join(args.argv);
}
```

```
from bcc import BPF, USDT
import sys

bpf_text = """
#include <uapi/linux/ptrace.h>

struct data_t {
    u64 timestamp;
    char client_ip;
    char file_path;
```

```
, char *ipstrptr, char *pathptr, u32 file_size) {
```

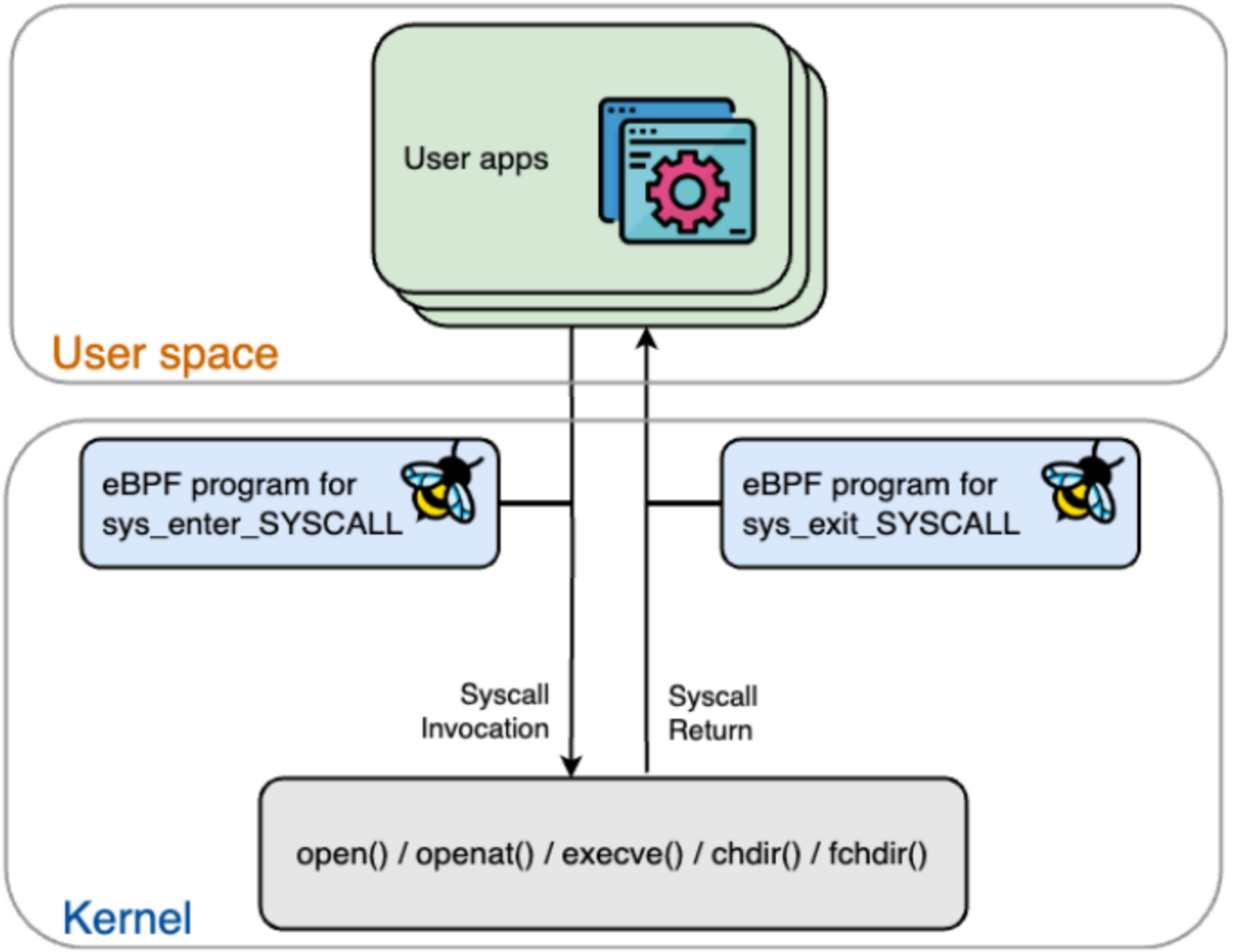
```
lata.client_ip), (void *)ipstrptr);
lata.file_path), (void *)pathptr);
ta));
```

```
u.enable_probe(probe="file_transfer", fn_name="trace_file_transfers")
b = BPF(text=bpf_text, usdt_contexts=[u])

def print_event(cpu, data, size):
    event = b["events"].event(data)
    print("{0}: {1} is downloading file {2} ({3} bytes)".format(
        event.timestamp, event.client_ip, event.file_path, event.file_size))

b["events"].open_perf_buffer(print_event)
while True:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()
```

What's YAULT?

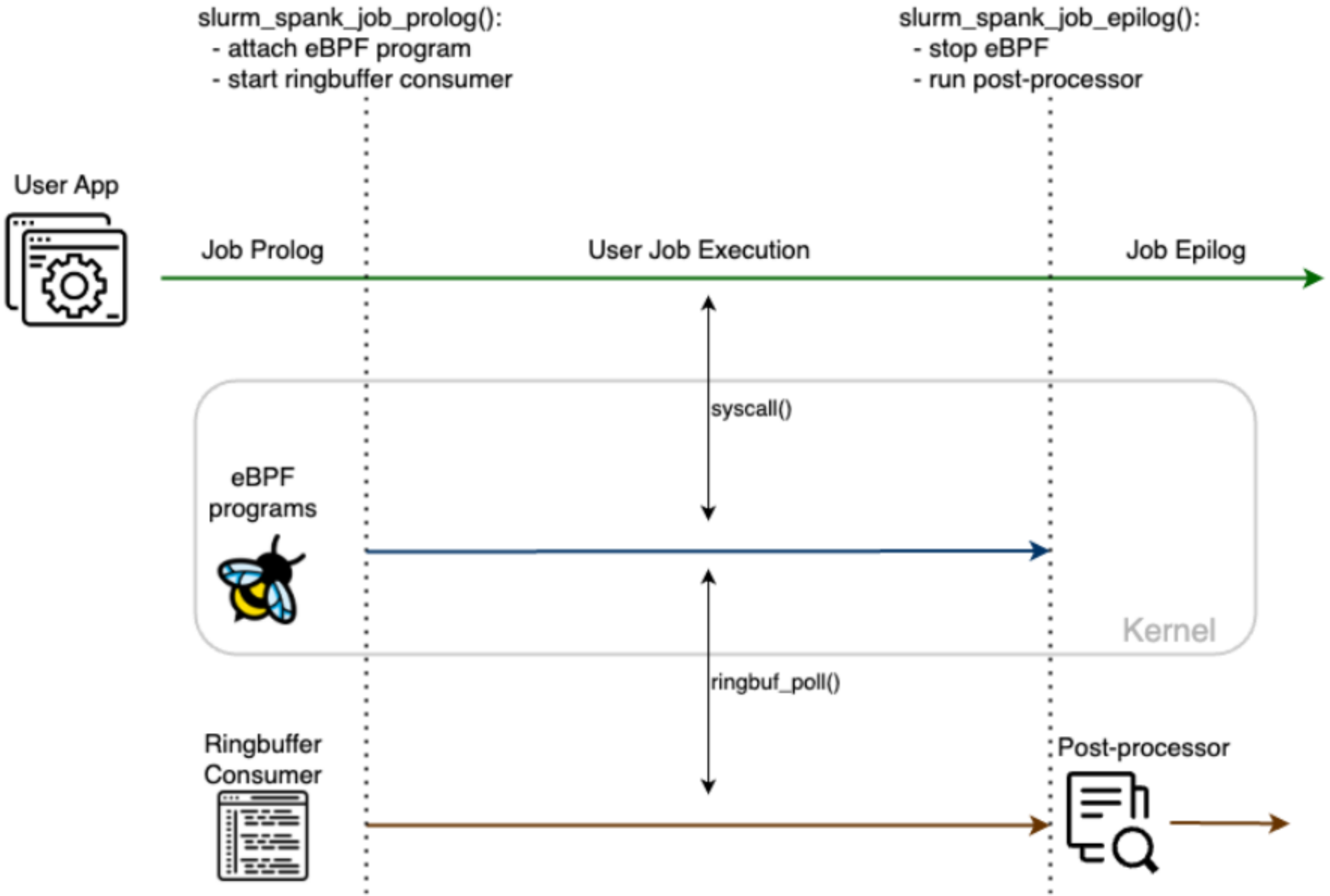
What are we thinking?

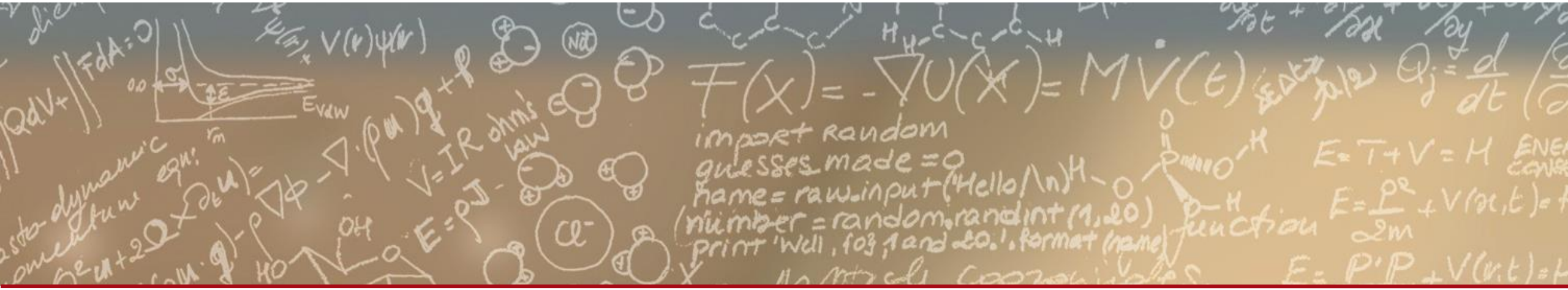
- We use eBPF to trace user application syscalls and their arguments
- Slurm plugin to associate application and Slurm job context
- Use this information to identify user application



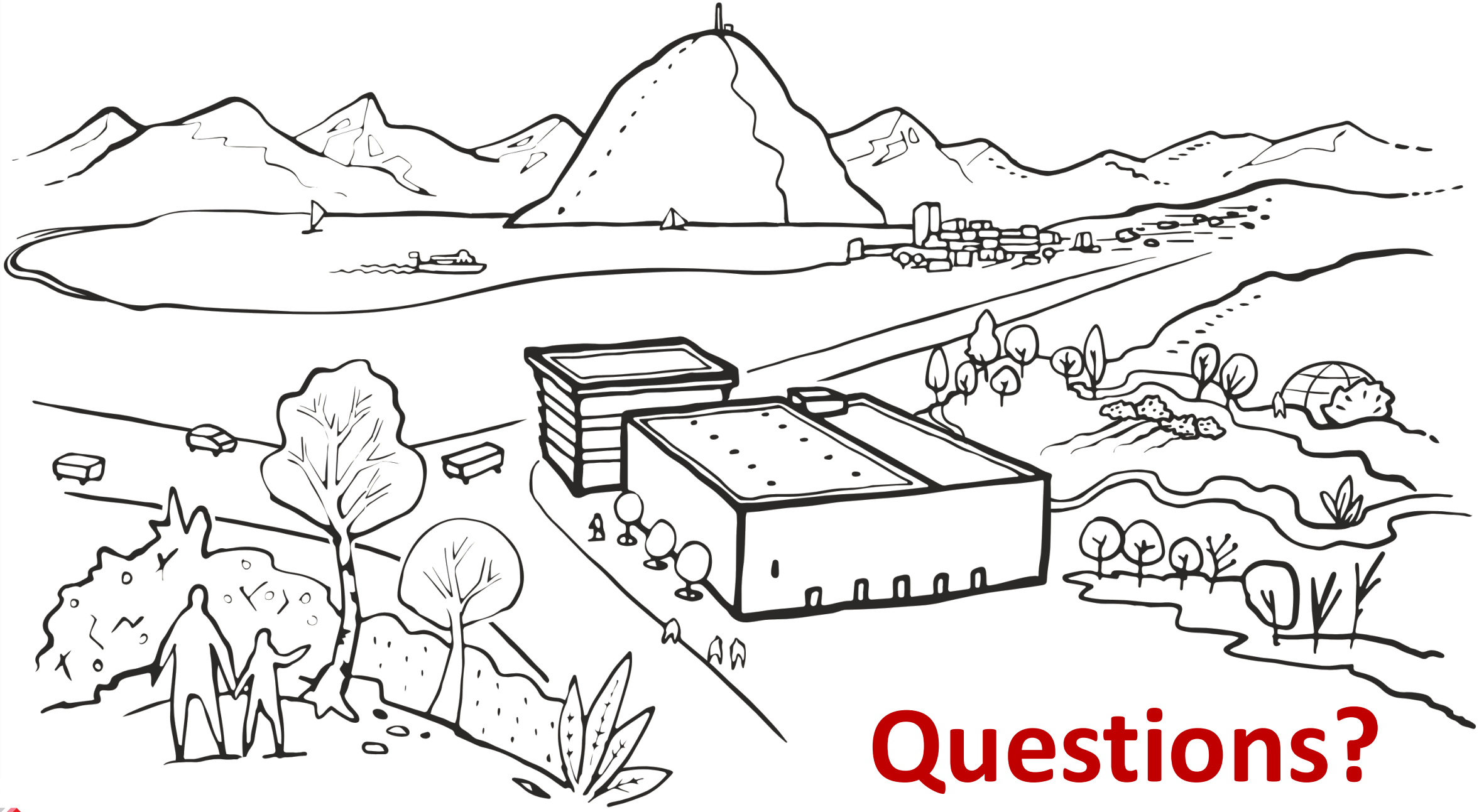
What's YAULT?

What are we thinking?





Thank you for your attention.



Questions?