Run Fast!

Catch Performance Regressions in eBPF with Rust

Everett Pompeii

everett@bencher.dev

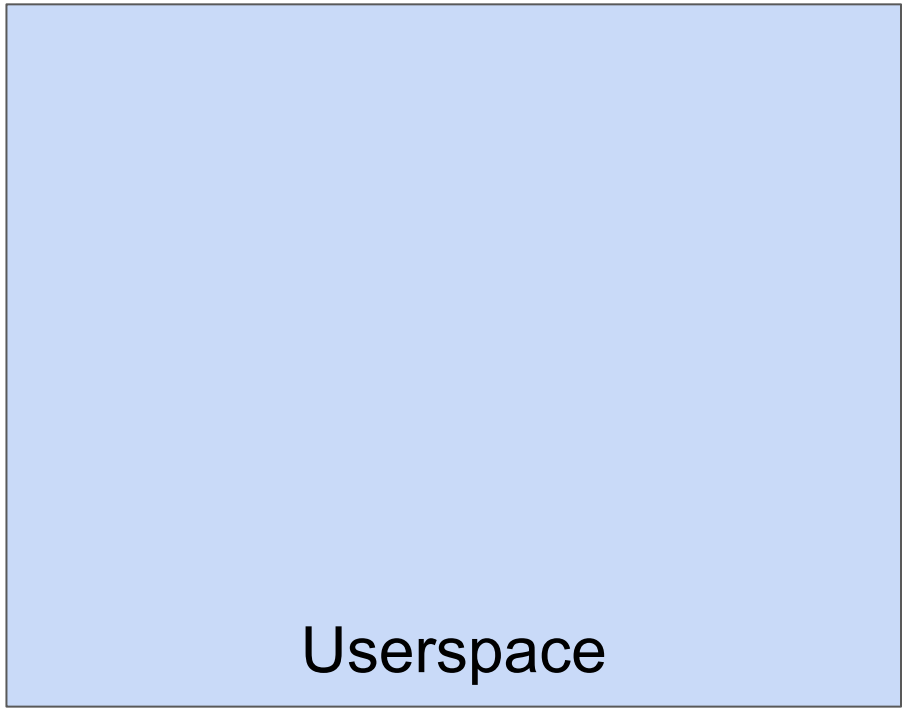
https://github.com/bencherdev/bencher

Bencher

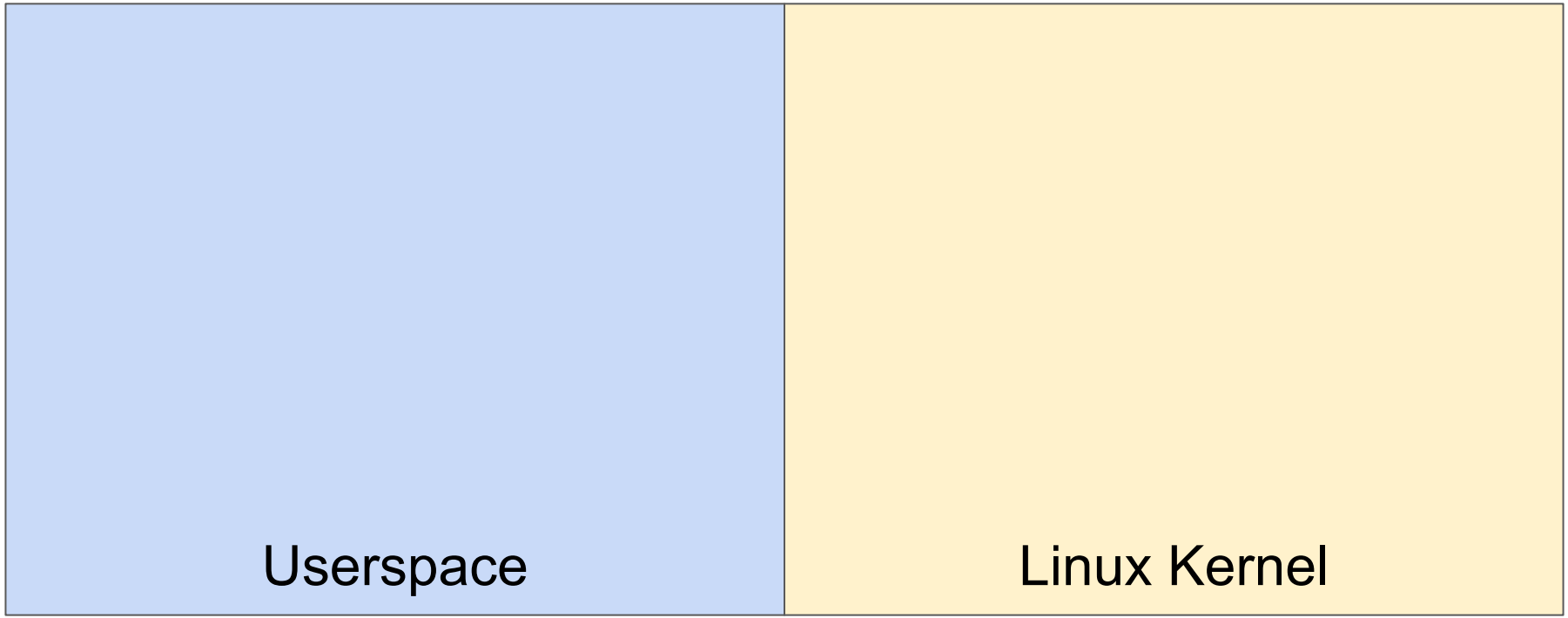
What is eBPF?

What is eBPF?

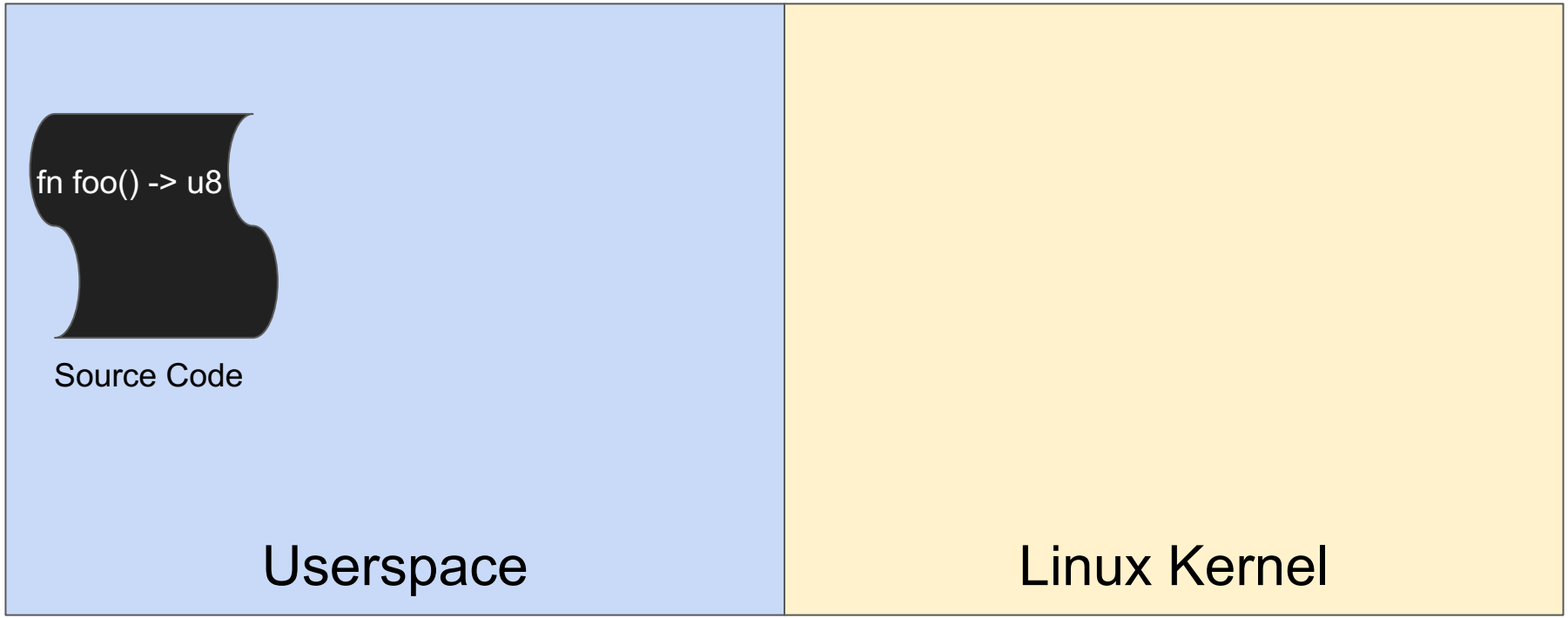
What is eBPF?



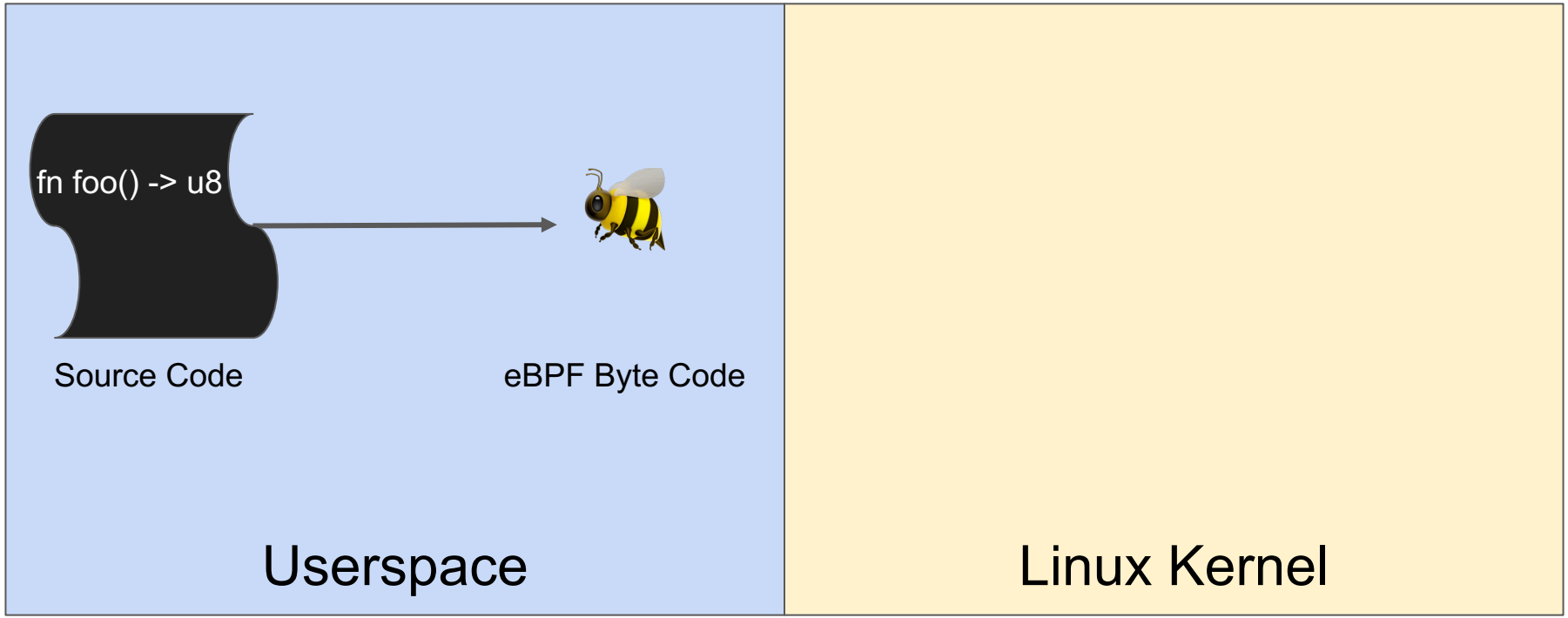
What is eBPF?



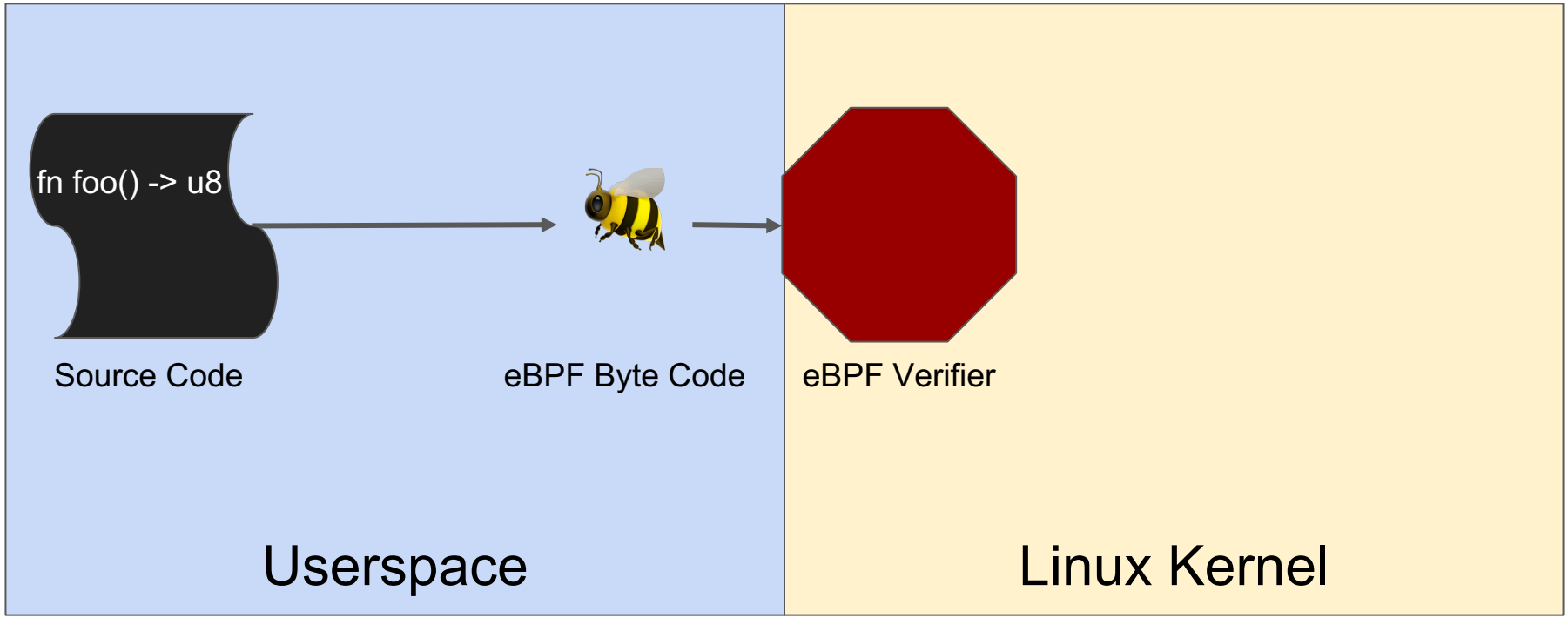
What is eBPF?



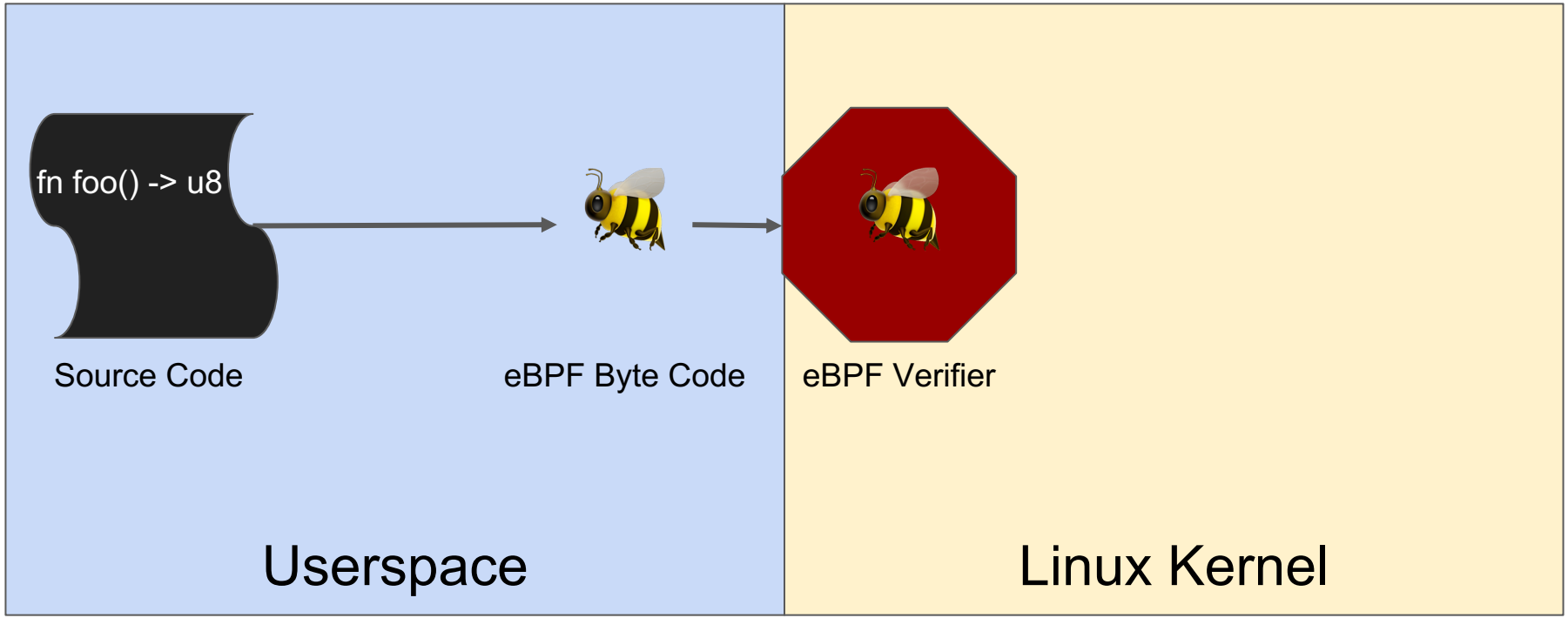
What is eBPF?



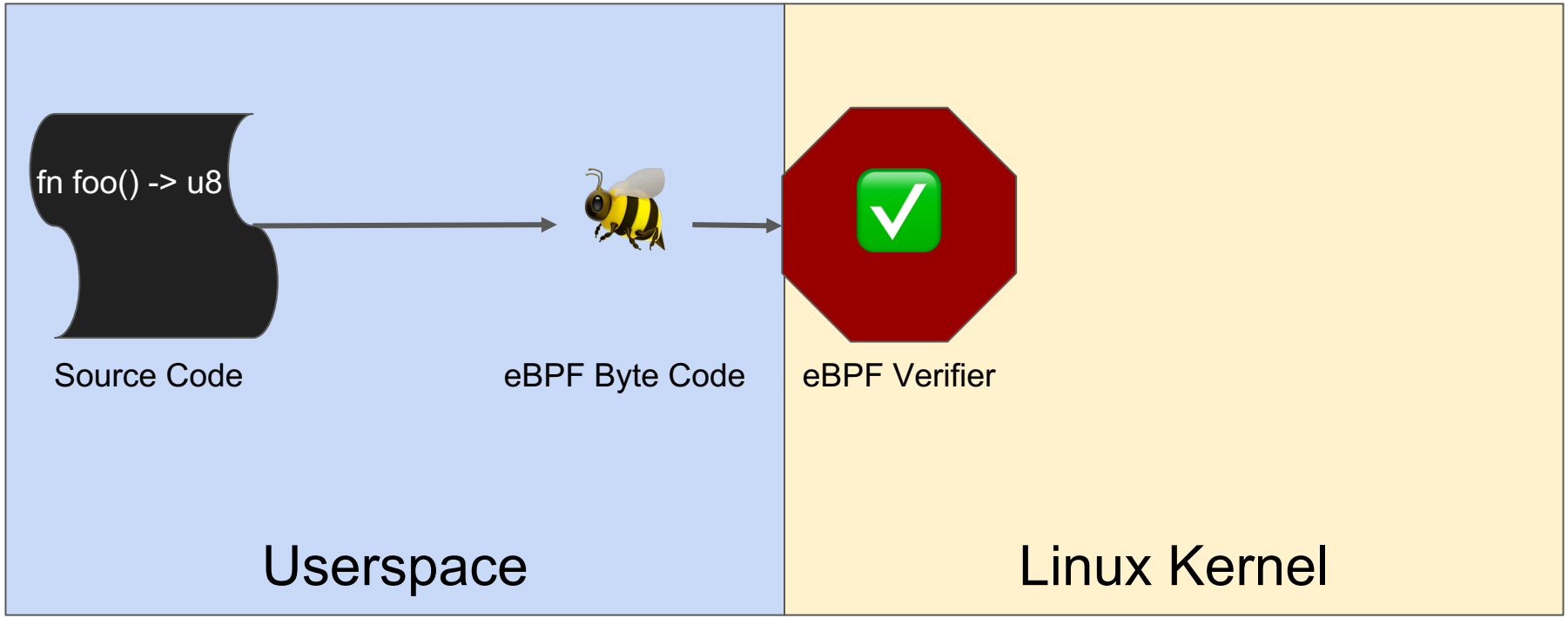
What is eBPF?


fn foo() -> u8
Source Code
eBPF Byte Code
eBPF Verifier
Userspace
Linux Kernel

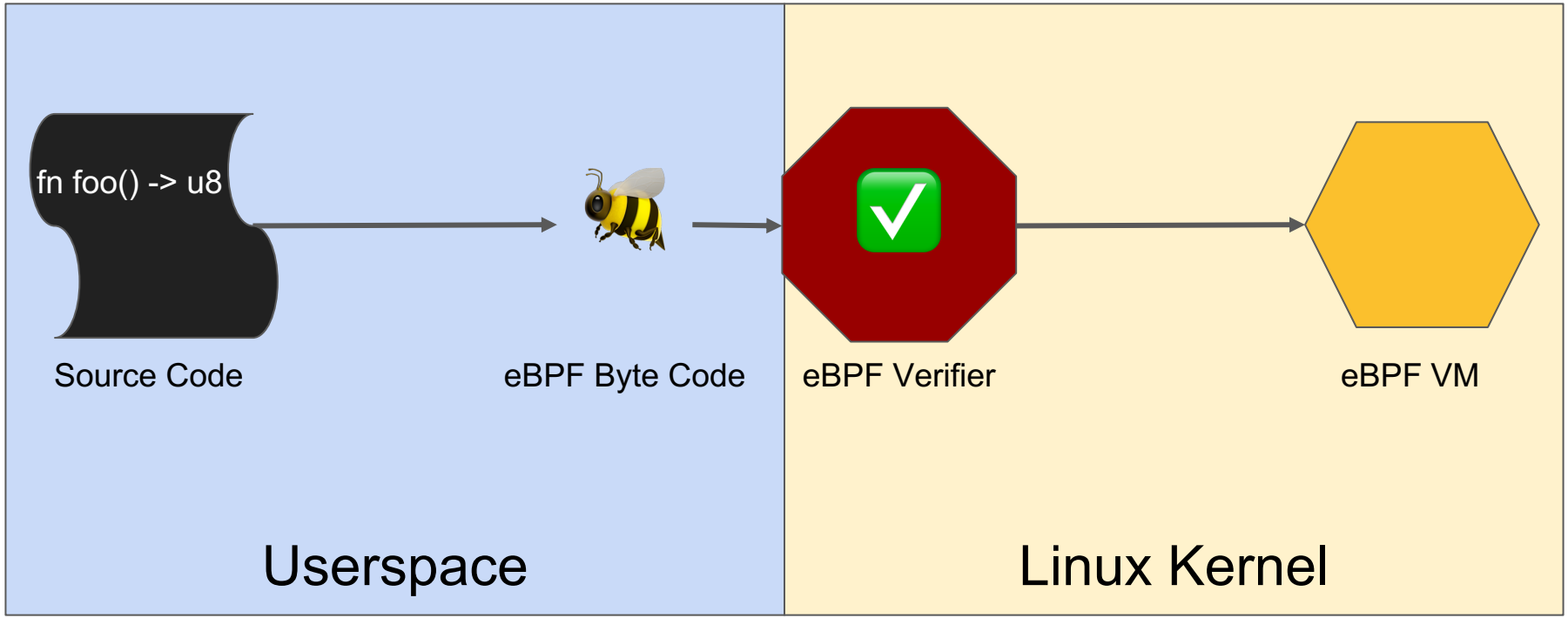
What is eBPF?


fn foo() -> u8
Source Code
eBPF Byte Code
eBPF Verifier
Userspace
Linux Kernel

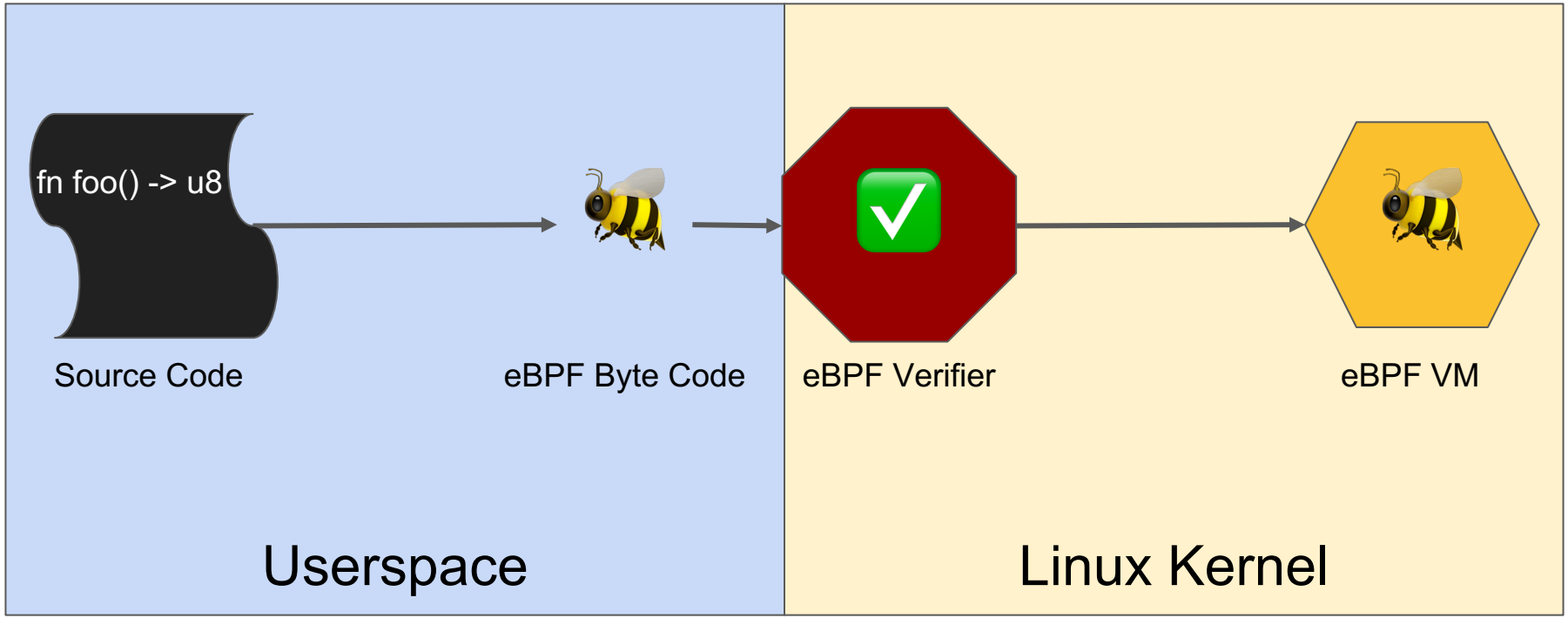
What is eBPF?


fn foo() -> u8
Source Code
eBPF Byte Code
eBPF Verifier
Userspace
Linux Kernel

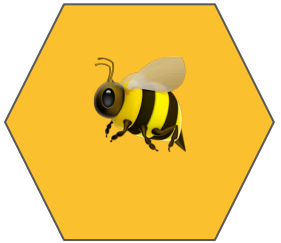
What is eBPF?


fn foo() -> u8
Source Code
eBPF Byte Code
eBPF Verifier
eBPF VM
Userspace
Linux Kernel

What is eBPF?


fn foo() -> u8
Source Code
eBPF Byte Code
eBPF Verifier
eBPF VM
Userspace
Linux Kernel

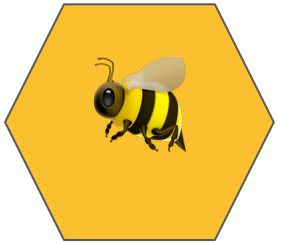
What is eBPF?



eBPF VM

What is eBPF?

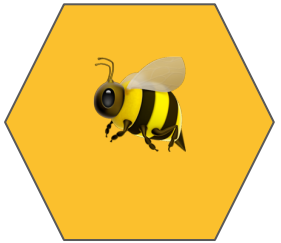
- tracepoint: Trace Syscalls



eBPF VM

What is eBPF?

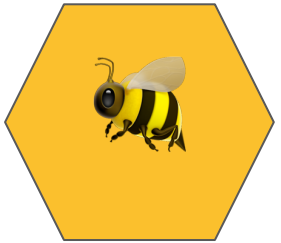
- tracepoint: Trace Syscalls
- uprobe: Userspace Probe



eBPF VM

What is eBPF?

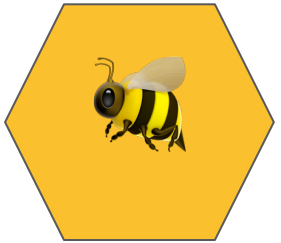
- tracepoint: Trace Syscalls
- uprobe: Userspace Probe
- kprobe: Kernel Function Probe



eBPF VM

What is eBPF?

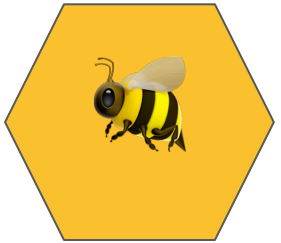
- tracepoint: Trace Syscalls
- uprobe: Userspace Probe
- kprobe: Kernel Function Probe
- xdp: Packet Filtering



eBPF VM

What is eBPF?

- tracepoint: Trace Syscalls
- uprobe: Userspace Probe
- kprobe: Kernel Function Probe
- xdp: Packet Filtering
- lsm: Linux Security Module

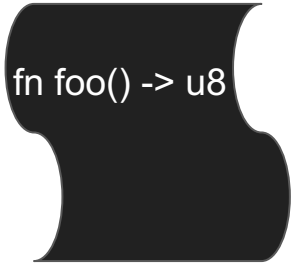


eBPF VM

What is Rust?

What is Rust?

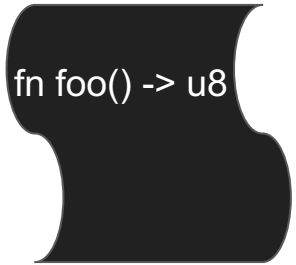
What is Rust?



Source Code

What is Rust?

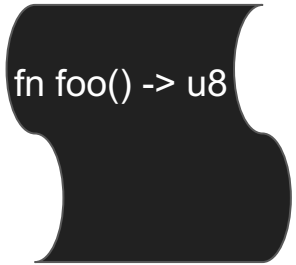
- Performance



Source Code

What is Rust?

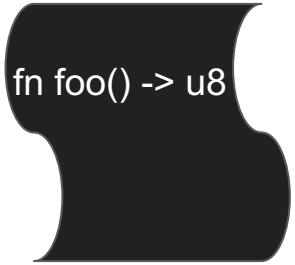
- Performance
- Reliability



Source Code

What is Rust?

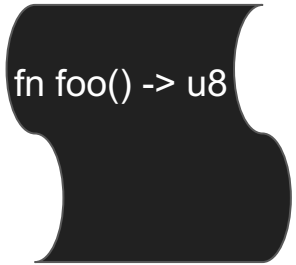
- Performance
- Reliability
- Productivity



Source Code

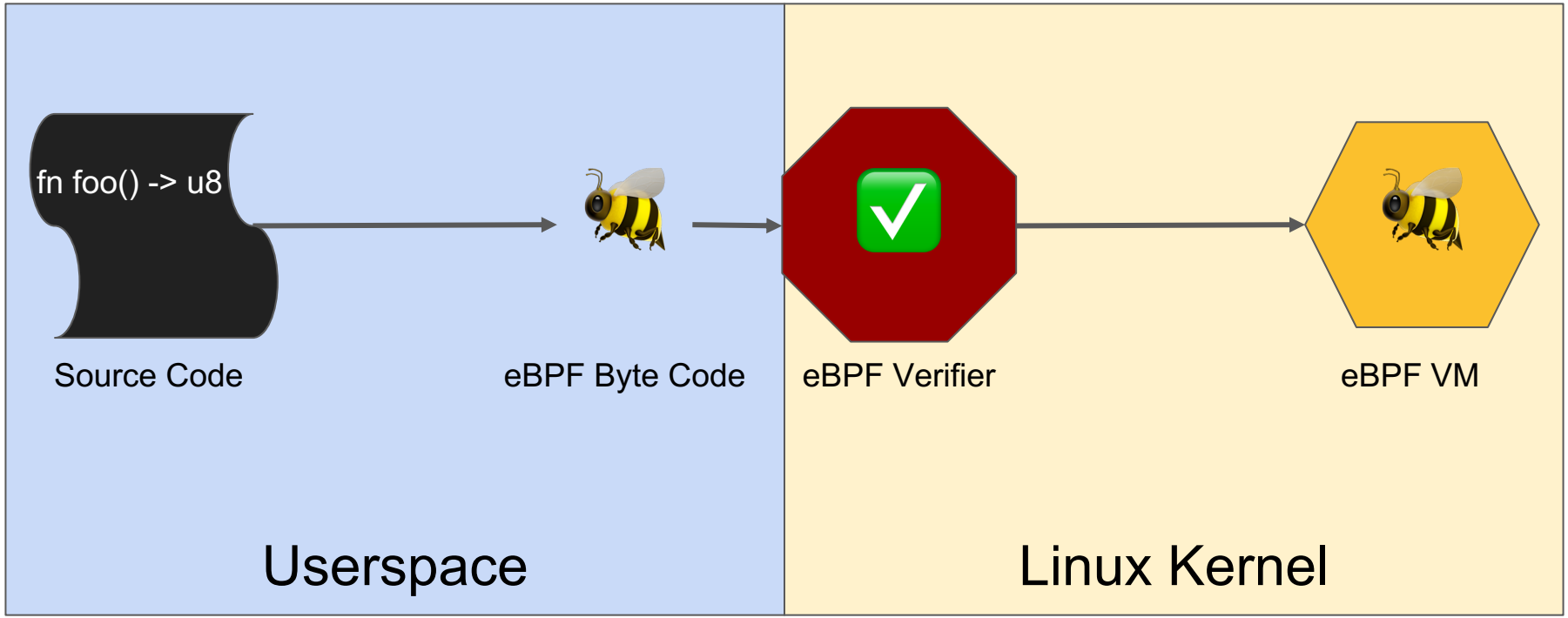
What is Rust?

- Performance
- Reliability
- Productivity

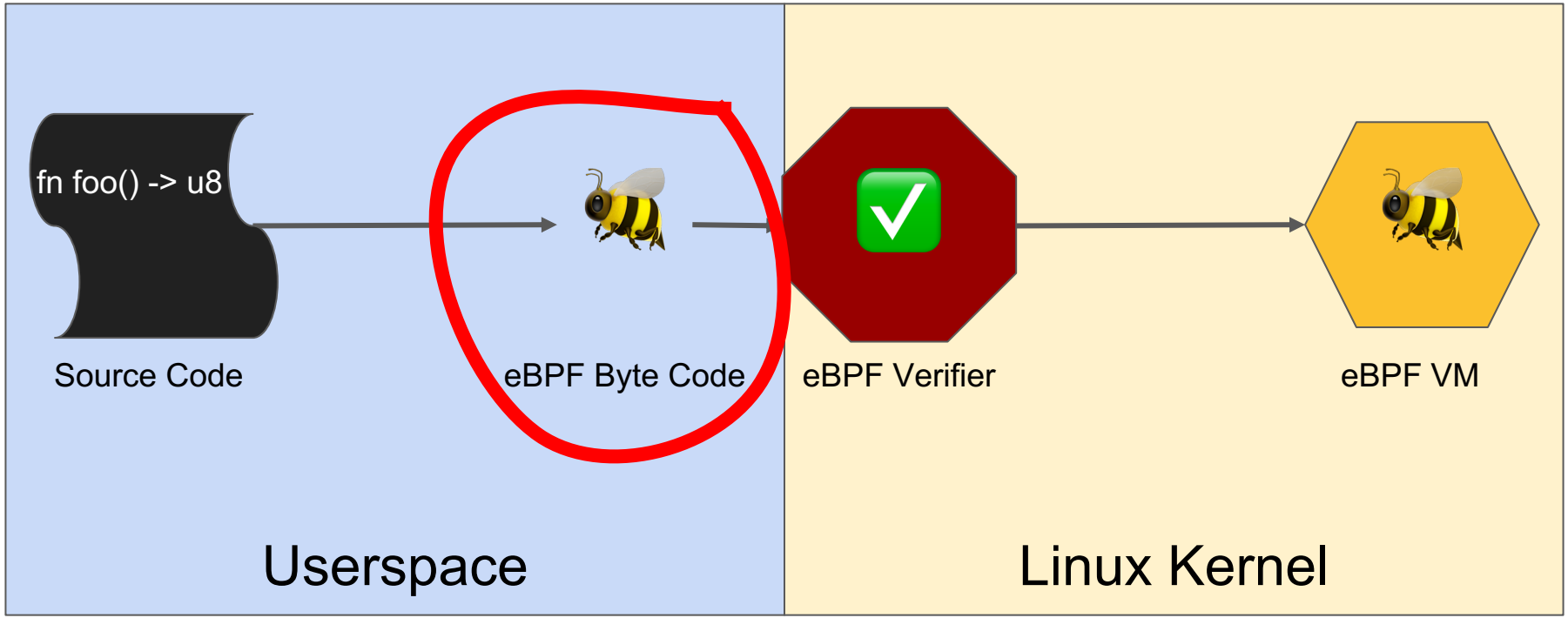


Source Code

What is Rust?


fn foo() -> u8
Source Code
eBPF Byte Code
eBPF Verifier
eBPF VM
Userspace
Linux Kernel

What is Rust?


fn foo() -> u8
Source Code
eBPF Byte Code
eBPF Verifier
eBPF VM
Userspace
Linux Kernel

What is a Performance Regression?

Performance bugs are bugs

eBPF: Up to 100ms of latency

Production: The most costly place to find bugs

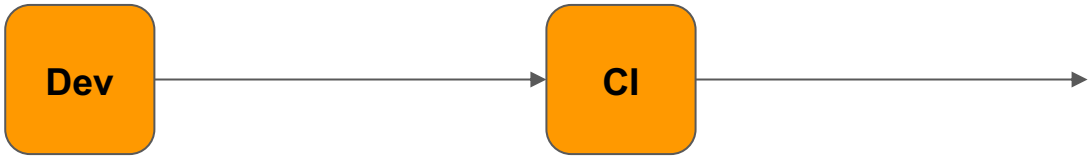
When do performance regression get detected?

When do performance regression get detected?

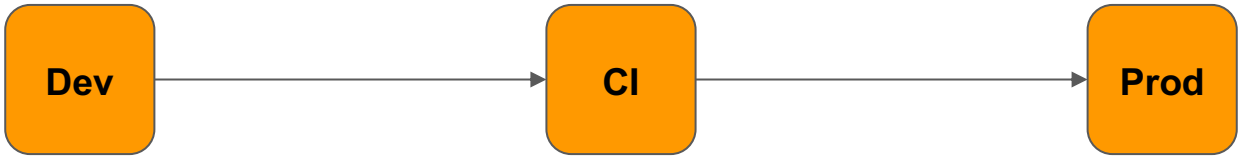
When do performance regression get detected?

When do performance regression get detected?

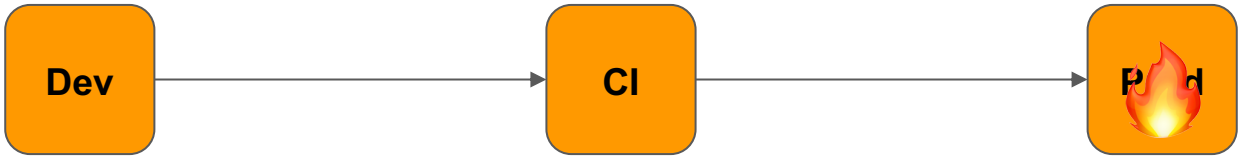
When do performance regression get detected?



When do performance regression get detected?



When do performance regression get detected?



Overview

Overview

- Basic eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust
- Benchmark eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust
- Benchmark eBPF program in Rust
- Continuous Benchmarking eBPF program in Rust

eBPF Tooling

eBPF Tooling

C

eBPF Tooling

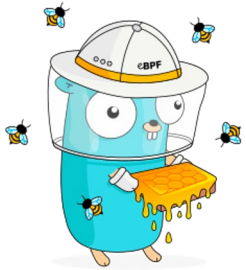
C

C/Python/Lua

eBPF Tooling

C

C/Python/Lua



Go

eBPF Tooling

C

C/Python/Lua

Go

Rust

eBPF Rust Tooling

Library	Userspace	eBPF	Syscalls

eBPF Rust Tooling

- libbpf-rs: Minimal and opinionated eBPF tooling for the Rust ecosystem

Library	Userspace	eBPF	Syscalls
libbpf-rs	🦀 Rust	🪚 C	🪚 C (libbpf)

eBPF Rust Tooling

- libbpf-rs: Minimal and opinionated eBPF tooling for the Rust ecosystem
- RedBPF: Rust library for building and running BPF/eBPF modules

Library	Userspace	eBPF	Syscalls
libbpf-rs	🦀 Rust	📦 C	📦 C (libbpf)
RedBPF	🦀 Rust	🦀 Rust	📦 C (libbpf)

eBPF Rust Tooling

- libbpf-rs: Minimal and opinionated eBPF tooling for the Rust ecosystem
- RedBPF: Rust library for building and running BPF/eBPF modules
- Aya: An eBPF library for the Rust programming language, built with a focus on developer experience and operability

Library	**Userspace**	**eBPF**	**Syscalls**
libbpf-rs	🦀 Rust	📦 C	📦 C (libbpf)
RedBPF	🦀 Rust	🦀 Rust	📦 C (libbpf)
Aya	🦀 Rust	🦀 Rust	🦀 Rust

Overview

Overview

- Basic eBPF program in Rust

eBPF XDP

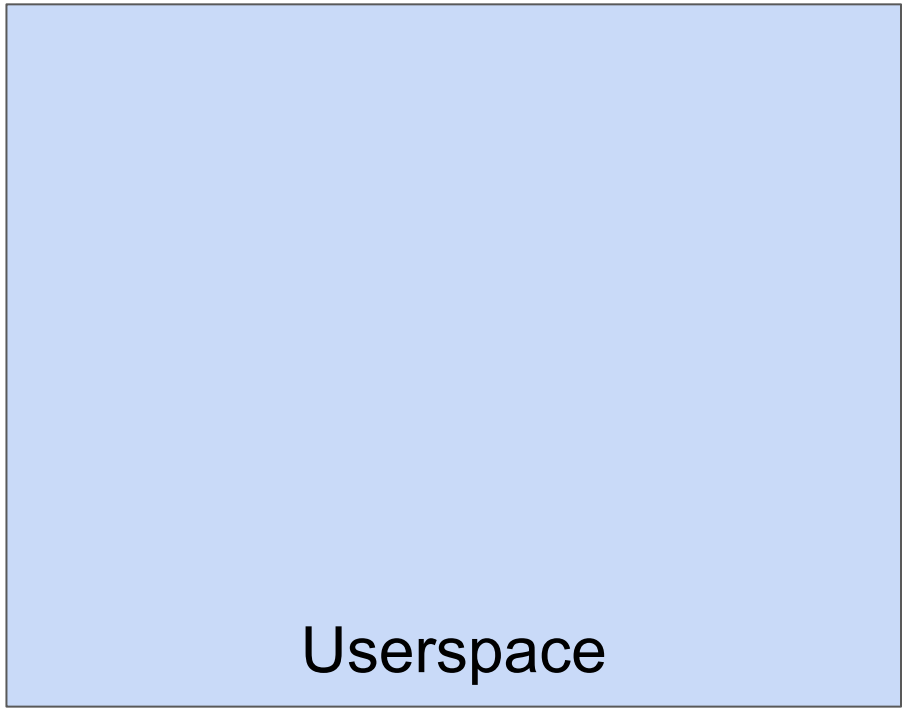
App v0: eBPF XDP

App v0: eBPF XDP

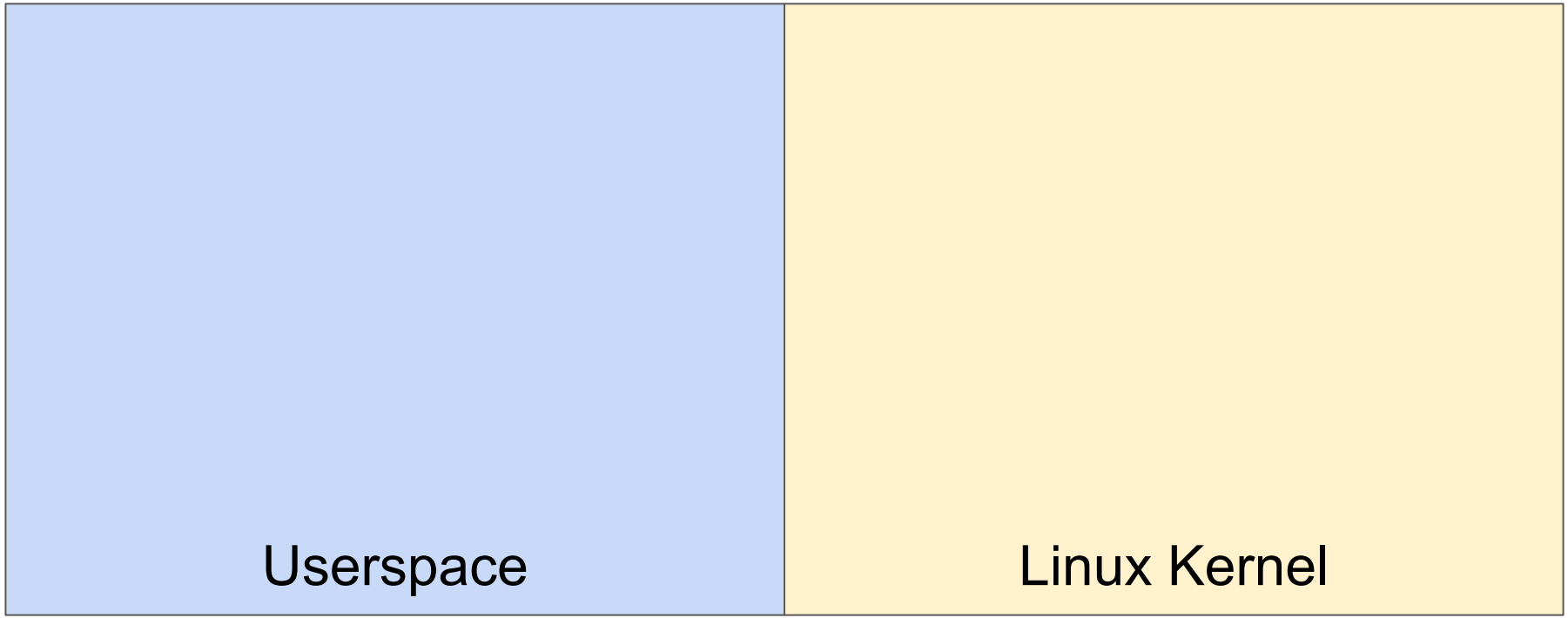
Log the IPv4 source address.

App v0: eBPF XDP

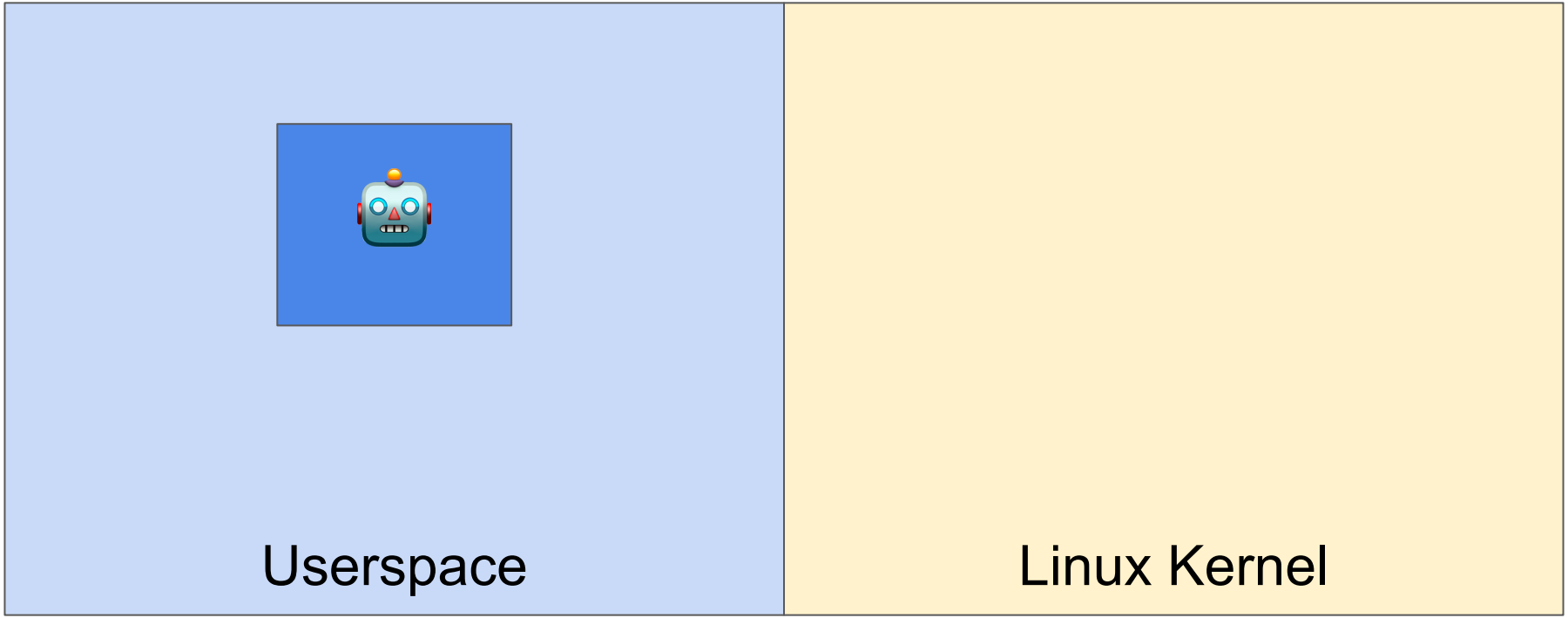
App v0: eBPF XDP



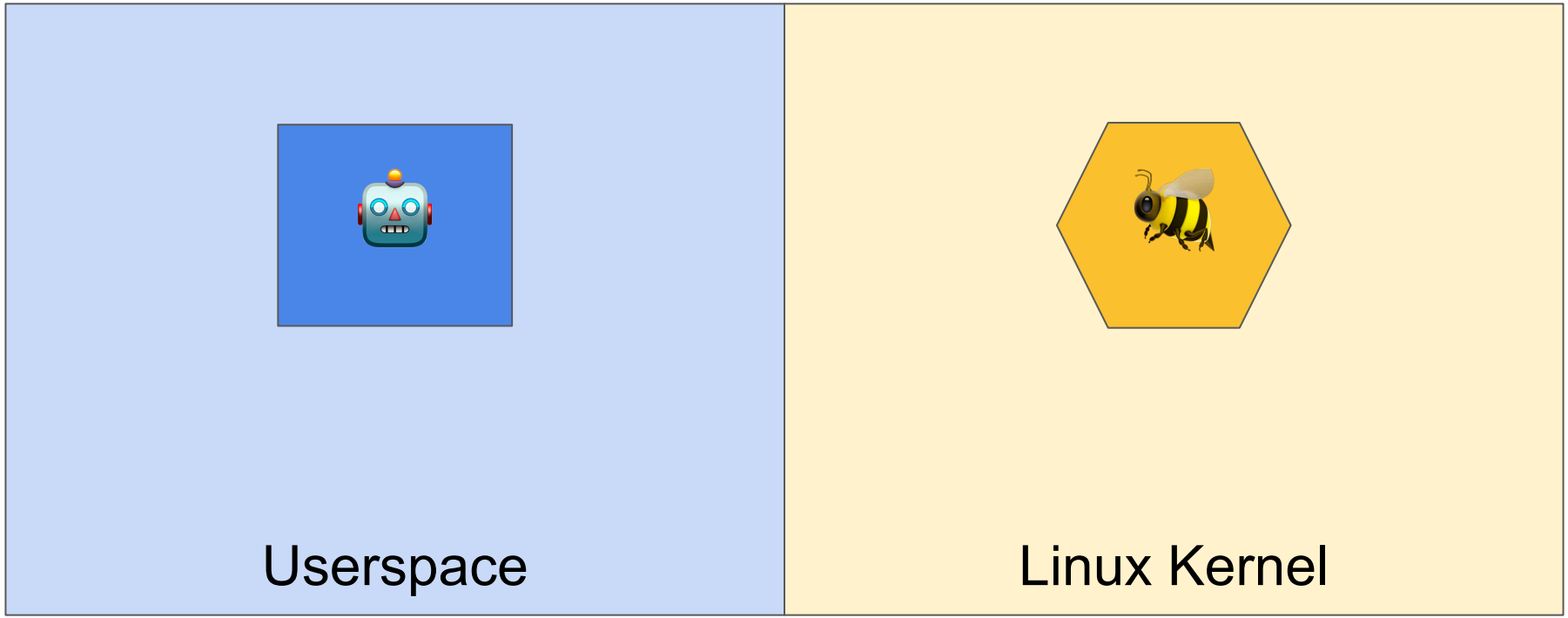
App v0: eBPF XDP



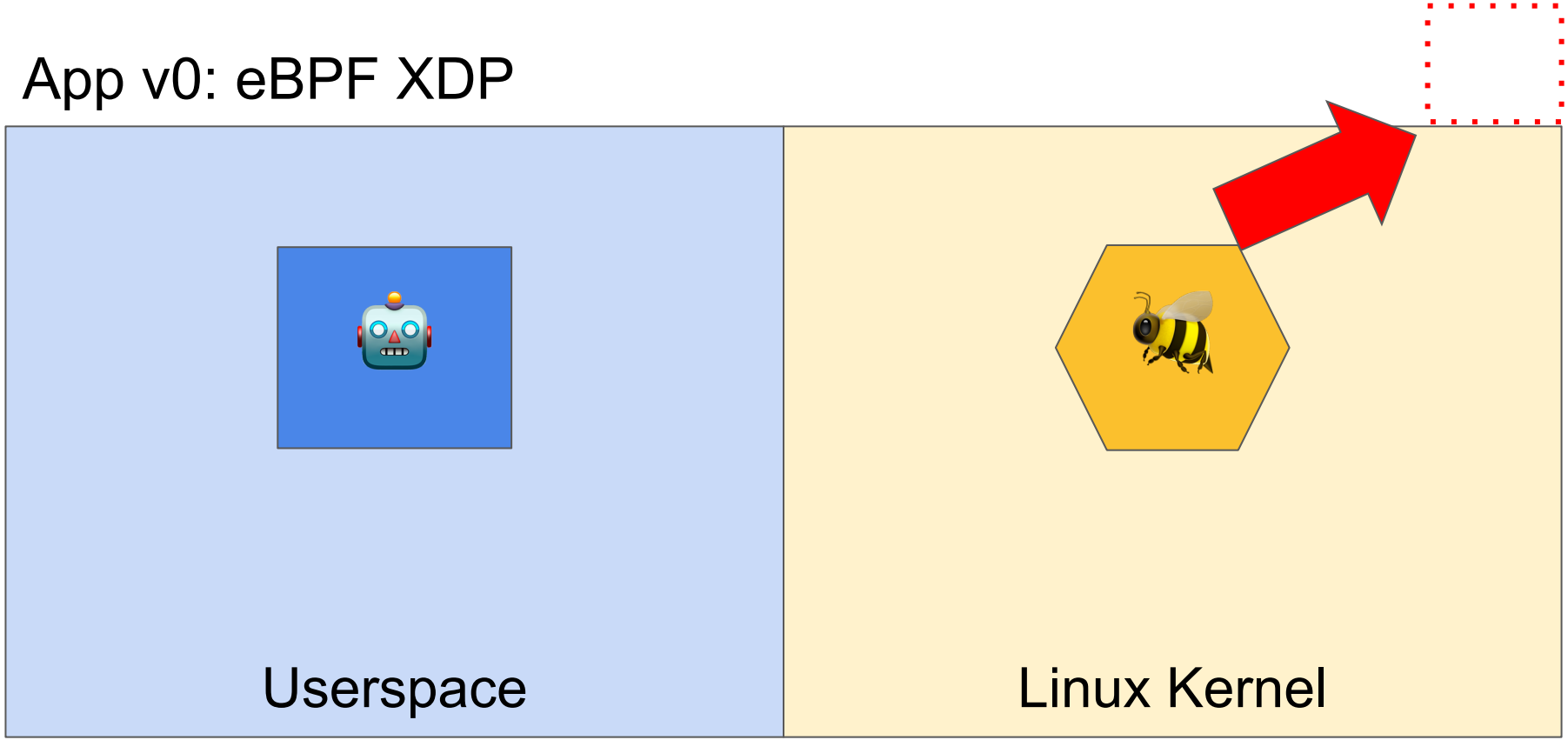
App v0: eBPF XDP



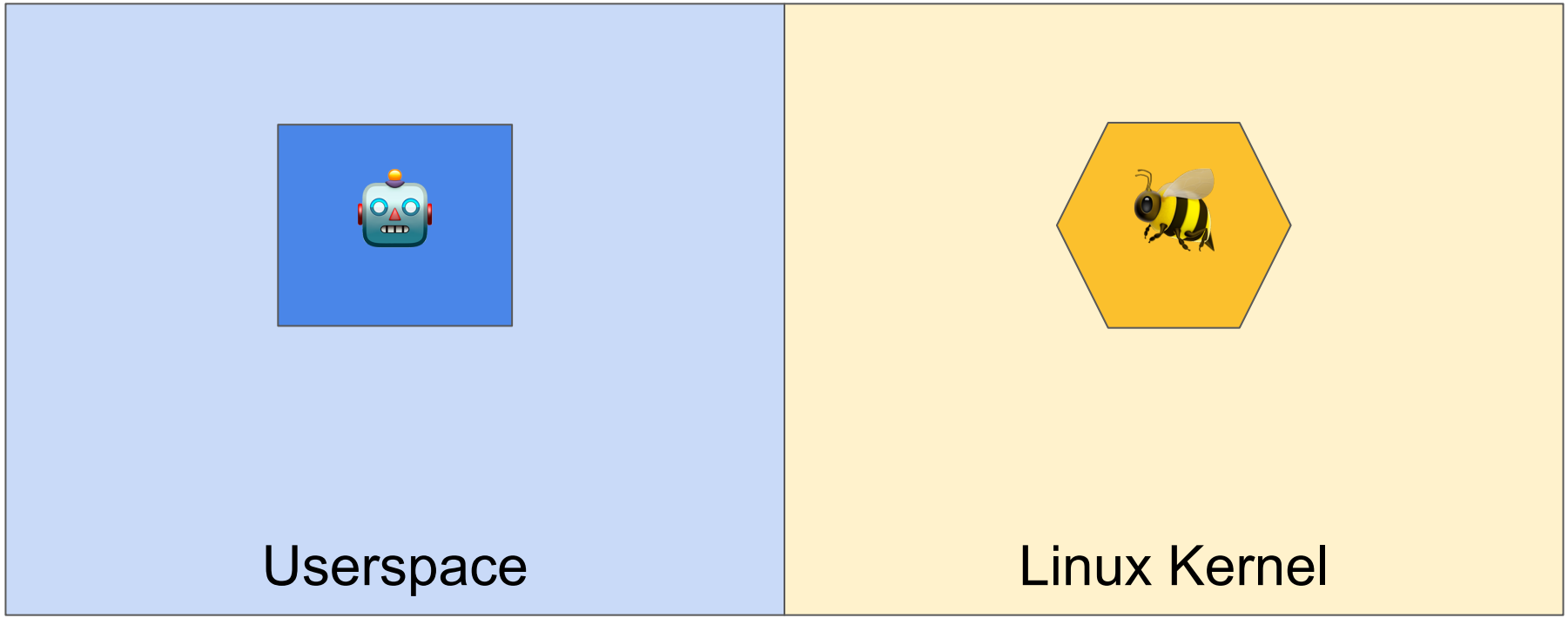
App v0: eBPF XDP



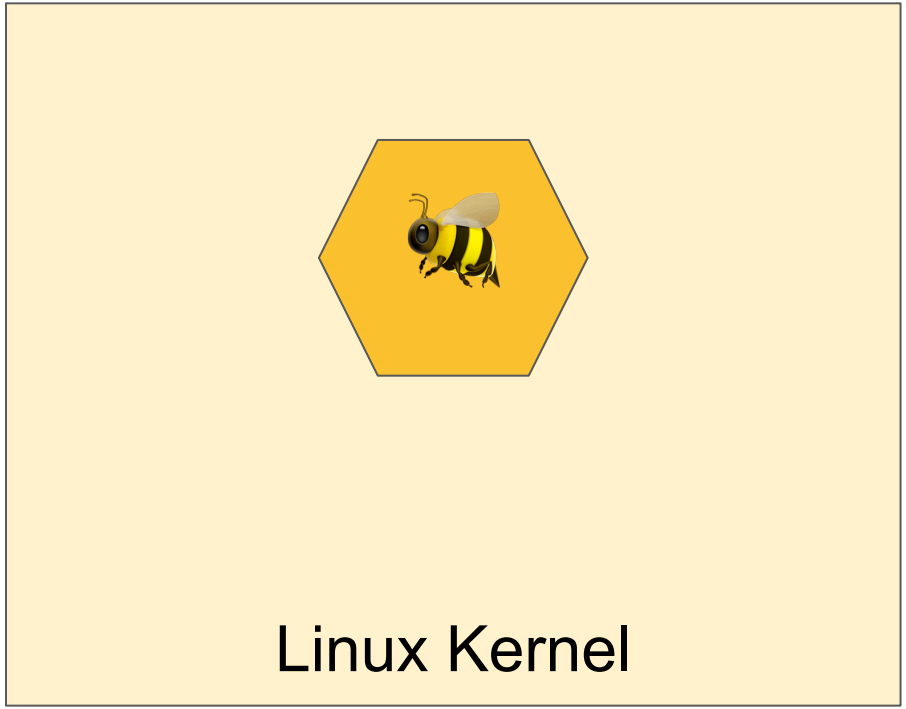
App v0: eBPF XDP



App v0: eBPF XDP



App v0: eBPF XDP



App v0: eBPF XDP

App v0: eBPF XDP

```
#[xdp(name = "fun_xdp")]
pub fn fun_xdp(ctx: XdpContext) -> u32 {
    match try_fun_xdp(ctx) {
        Ok(ret) => ret,
        Err(_) => xdp_action::XDP_ABORTED,
    }
}
```

App v0: eBPF XDP

```
#[xdp(name = "fun_xdp")]
pub fn fun_xdp(ctx: XdpContext) -> u32 {




    ,
}
```

App v0: eBPF XDP

```rust
#[xdp(name = "fun_xdp")]
pub fn fun_xdp(ctx: XdpContext) -> u32 {
    match try_fun_xdp(&ctx) {



    }
}
```

App v0: eBPF XDP

```rust
#[xdp(name = "fun_xdp")]
pub fn fun_xdp(ctx: XdpContext) -> u32 {
    match try_fun_xdp(&ctx) {
        Ok(ret) => ret,

    }
}
```

App v0: eBPF XDP

```rust
#[xdp(name = "fun_xdp")]
pub fn fun_xdp(ctx: XdpContext) -> u32 {
    match try_fun_xdp(&ctx) {
        Ok(ret) => ret,
        Err(_) => xdp_action::XDP_ABORTED,
    }
}
```

App v0: eBPF XDP

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {

}
```

App v0: eBPF XDP

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    let eth_hdr: *const EthHdr = unsafe { ptr_at(ctx, 0)? };









}
```

App v0: eBPF XDP

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    let eth_hdr: *const EthHdr = unsafe { ptr_at(ctx, 0)? };
    unsafe {
        let EtherType::Ipv4 = (*eth_hdr).ether_type else {
            return Ok(xdp_action::XDP_PASS);
        };
    }




}
```

App v0: eBPF XDP

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    let eth_hdr: *const EthHdr = unsafe { ptr_at(ctx, 0)? };
    unsafe {
        let EtherType::Ipv4 = (*eth_hdr).ether_type else {
            return Ok(xdp_action::XDP_PASS);
        };
    }
    let ipv4_hdr: *const Ipv4Hdr = unsafe { ptr_at(ctx, EthHdr::LEN)? };









}
```

App v0: eBPF XDP

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    let eth_hdr: *const EthHdr = unsafe { ptr_at(ctx, 0)? };
    unsafe {
        let EtherType::Ipv4 = (*eth_hdr).ether_type else {
            return Ok(xdp_action::XDP_PASS);
        };
    }
    let ipv4_hdr: *const Ipv4Hdr = unsafe { ptr_at(ctx, EthHdr::LEN)? };
    let source_addr = unsafe { (*ipv4_hdr).src_addr };




}
```

App v0: eBPF XDP

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    let eth_hdr: *const EthHdr = unsafe { ptr_at(ctx, 0)? };
    unsafe {
        let EtherType::Ipv4 = (*eth_hdr).ether_type else {
            return Ok(xdp_action::XDP_PASS);
        };
    }
    let ipv4_hdr: *const Ipv4Hdr = unsafe { ptr_at(ctx, EthHdr::LEN)? };
    let source_addr = unsafe { (*ipv4_hdr).src_addr };
    info!(ctx, "IPv4 Source Address: {}", source_addr);


}
```

App v0: eBPF XDP

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    let eth_hdr: *const EthHdr = unsafe { ptr_at(ctx, 0)? };
    unsafe {
        let EtherType::Ipv4 = (*eth_hdr).ether_type else {
            return Ok(xdp_action::XDP_PASS);
        };
    }
    let ipv4_hdr: *const Ipv4Hdr = unsafe { ptr_at(ctx, EthHdr::LEN)? };
    let source_addr = unsafe { (*ipv4_hdr).src_addr };
    info!(ctx, "IPv4 Source Address: {}", source_addr);

    Ok(xdp_action::XDP_PASS)
}
```

App v0: eBPF XDP

```rust
#[inline(always)]
unsafe fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*const T, ()> {

}
```

App v0: eBPF XDP

```rust
#[inline(always)]
unsafe fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*const T, ()> {
    let start = ctx.data();
}
```

App v0: eBPF XDP

```rust
#[inline(always)]
unsafe fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*const T, ()> {
    let start = ctx.data();
    let end = ctx.data_end();

}
```

App v0: eBPF XDP

```rust
#[inline(always)]
unsafe fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*const T, ()> {
    let start = ctx.data();
    let end = ctx.data_end();
    let len = core::mem::size_of::<T>();

}
```

App v0: eBPF XDP

```rust
#[inline(always)]
unsafe fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*const T, ()> {
    let start = ctx.data();
    let end = ctx.data_end();
    let len = core::mem::size_of::<T>();

    if start + offset + len > end {
        return Err(());
    }

}
```

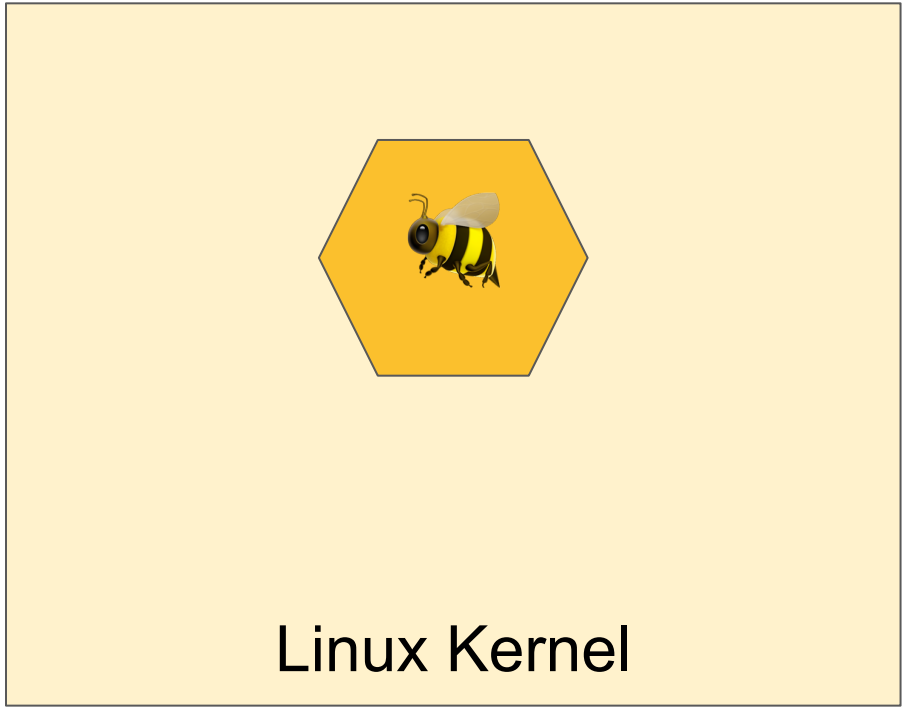
App v0: eBPF XDP

```rust
#[inline(always)]
unsafe fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*const T, ()> {
    let start = ctx.data();
    let end = ctx.data_end();
    let len = core::mem::size_of::<T>();

    if start + offset + len > end {
        return Err(());
    }

    Ok((start + offset) as _)
}
```

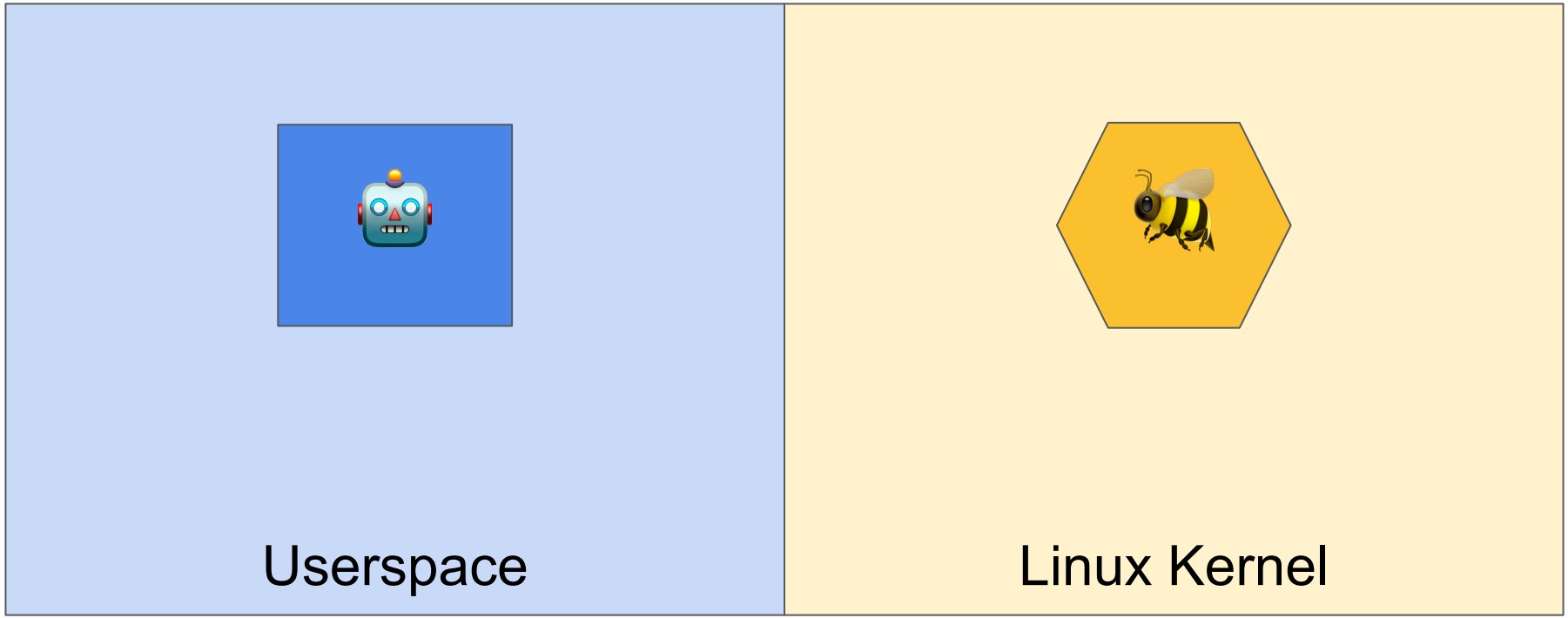
App v0: eBPF XDP

```rust
#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    unsafe { core::hint::unreachable_unchecked() }
}
```

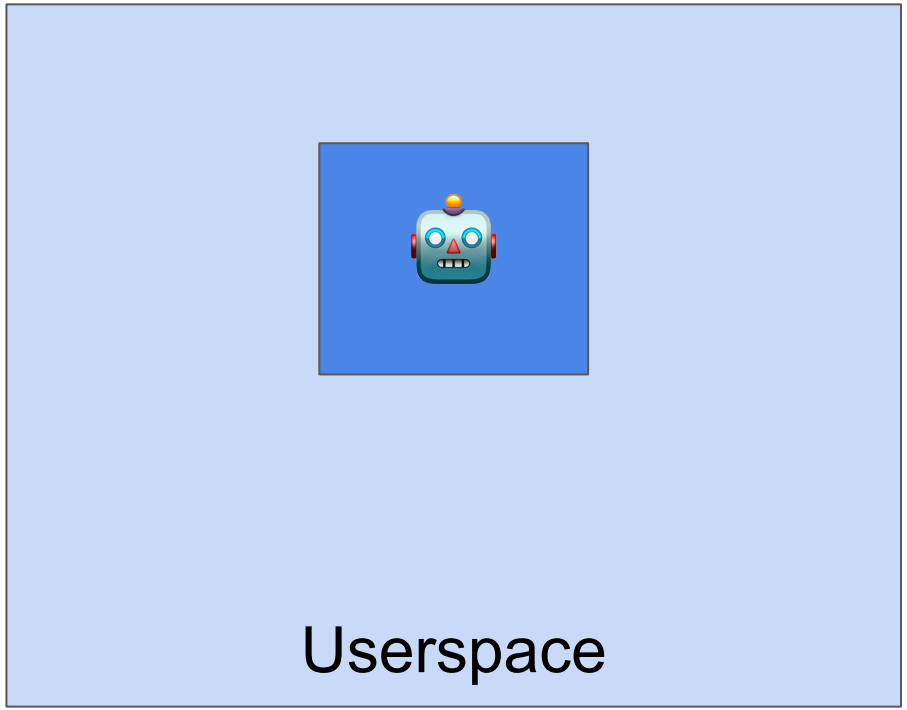
App v0: eBPF XDP



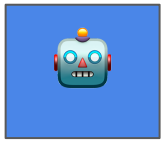
App v0: eBPF XDP



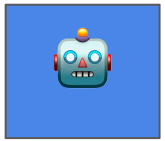
App v0: eBPF XDP



App v0: eBPF XDP

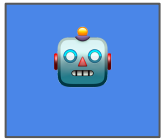


App v0: eBPF XDP



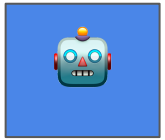
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {









}
```

App v0: eBPF XDP



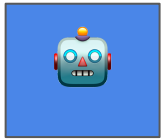
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();









}
```

App v0: eBPF XDP



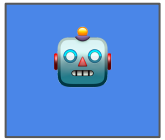
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();
    env_logger::init();









}
```

App v0: eBPF XDP



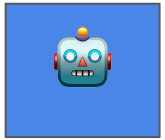
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();
    env_logger::init();
    let mut bpf = Bpf::load(include_bytes_aligned!("../path/to/ebpf-bin"))?;









}
```

App v0: eBPF XDP



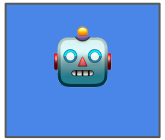
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();
    env_logger::init();
    let mut bpf = Bpf::load(include_bytes_aligned!("../path/to/ebpf-bin"))?;
    BpfLogger::init(&mut bpf)?;









}
```

App v0: eBPF XDP



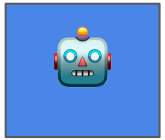
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();
    env_logger::init();
    let mut bpf = Bpf::load(include_bytes_aligned!("../path/to/ebpf-bin"))?;
    BpfLogger::init(&mut bpf)?;
    let program: &mut Xdp = bpf.program_mut("fun_xdp").unwrap().try_into()?;









}
```

App v0: eBPF XDP



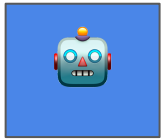
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();
    env_logger::init();
    let mut bpf = Bpf::load(include_bytes_aligned!("../path/to/ebpf-bin"))?;
    BpfLogger::init(&mut bpf)?;
    let program: &mut Xdp = bpf.program_mut("fun_xdp").unwrap().try_into()?;
    program.load()?;









}
```

App v0: eBPF XDP



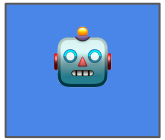
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();
    env_logger::init();
    let mut bpf = Bpf::load(include_bytes_aligned!("../path/to/ebpf-bin"))?;
    BpfLogger::init(&mut bpf)?;
    let program: &mut Xdp = bpf.program_mut("fun_xdp").unwrap().try_into()?;
    program.load()?;
    program.attach(&opt.iface, XdpFlags::default())?;




}
```

App v0: eBPF XDP



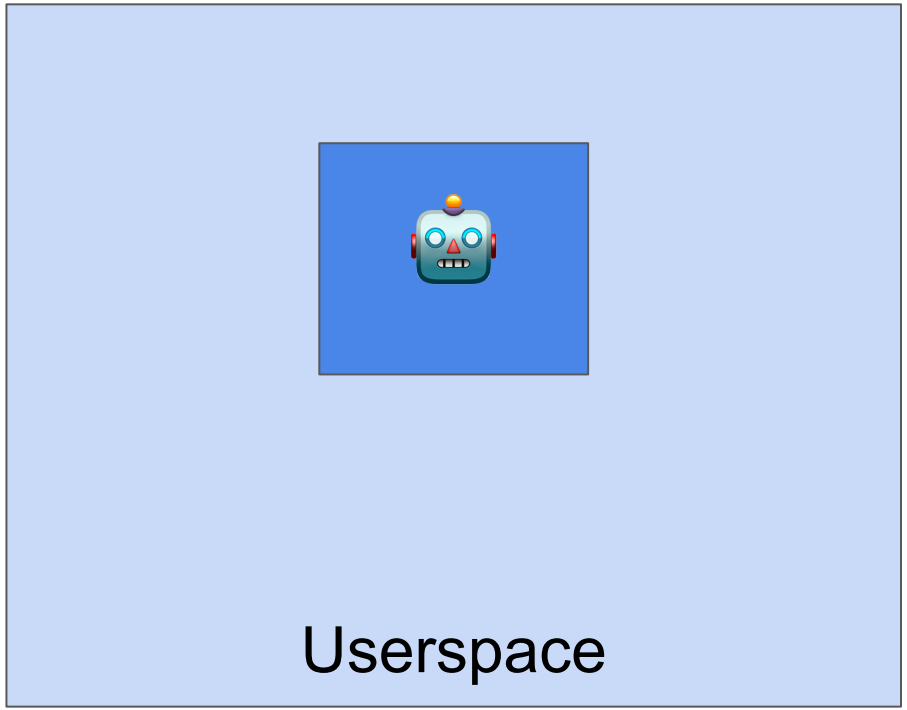
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();
    env_logger::init();
    let mut bpf = Bpf::load(include_bytes_aligned!("../path/to/ebpf-bin"))?;
    BpfLogger::init(&mut bpf)?;
    let program: &mut Xdp = bpf.program_mut("fun_xdp").unwrap().try_into()?;
    program.load()?;
    program.attach(&opt.iface, XdpFlags::default())?;

    info!("Waiting for Ctrl-C...");
    signal::ctrl_c().await?;


}
```

App v0: eBPF XDP

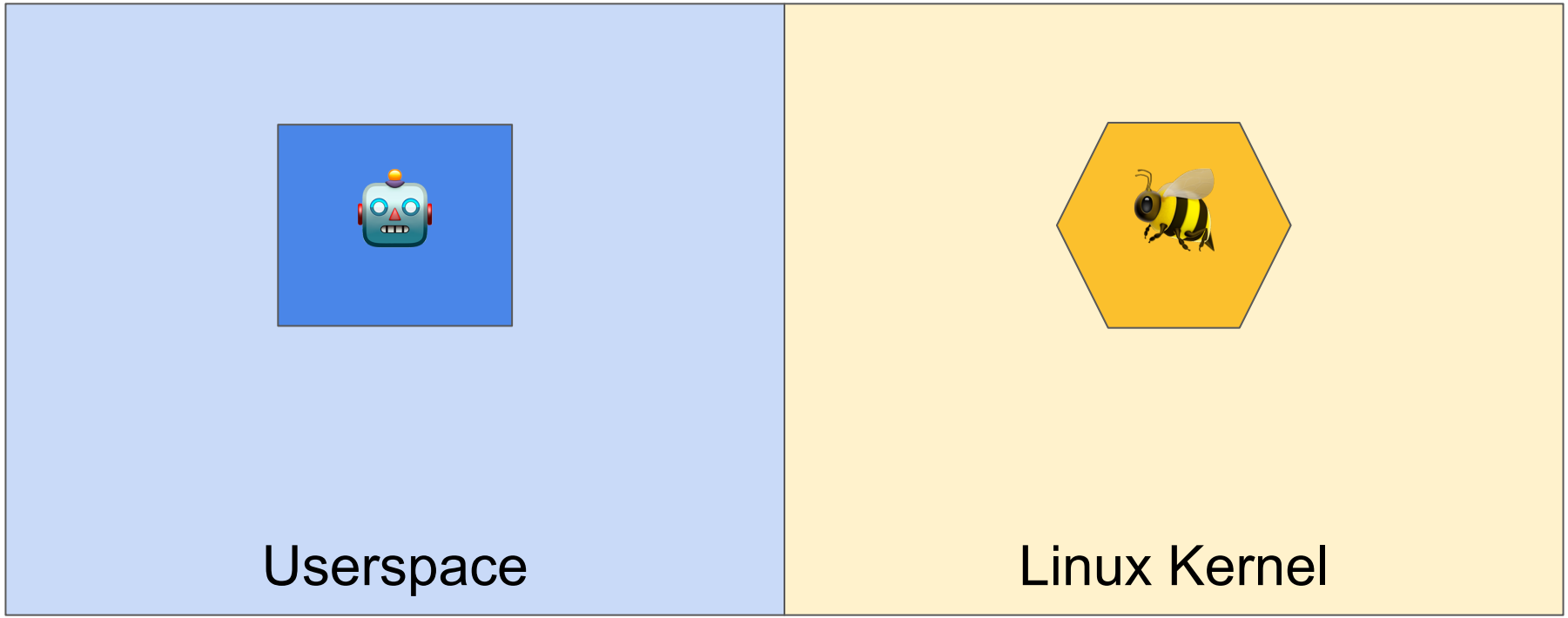


```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();
    env_logger::init();
    let mut bpf = Bpf::load(include_bytes_aligned!("../path/to/ebpf-bin"))?;
    BpfLogger::init(&mut bpf)?;
    let program: &mut Xdp = bpf.program_mut("fun_xdp").unwrap().try_into()?;
    program.load()?;
    program.attach(&opt.iface, XdpFlags::default())?;

    info!("Waiting for Ctrl-C...");
    signal::ctrl_c().await?;
    info!("Exiting...");
    Ok(())
}
```

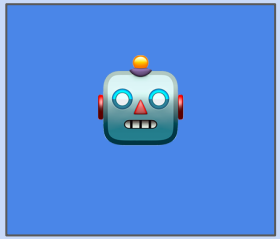
App v0: eBPF XDP

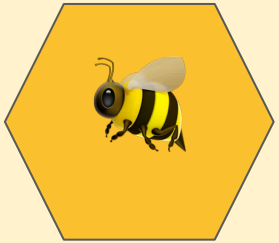


App v0: eBPF XDP



App v0: eBPF XDP





Userspace

Linux Kernel

Overview

- Basic eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust

eBPF Maps: userspace ⇔ eBPF

App v1: Fizz Feature

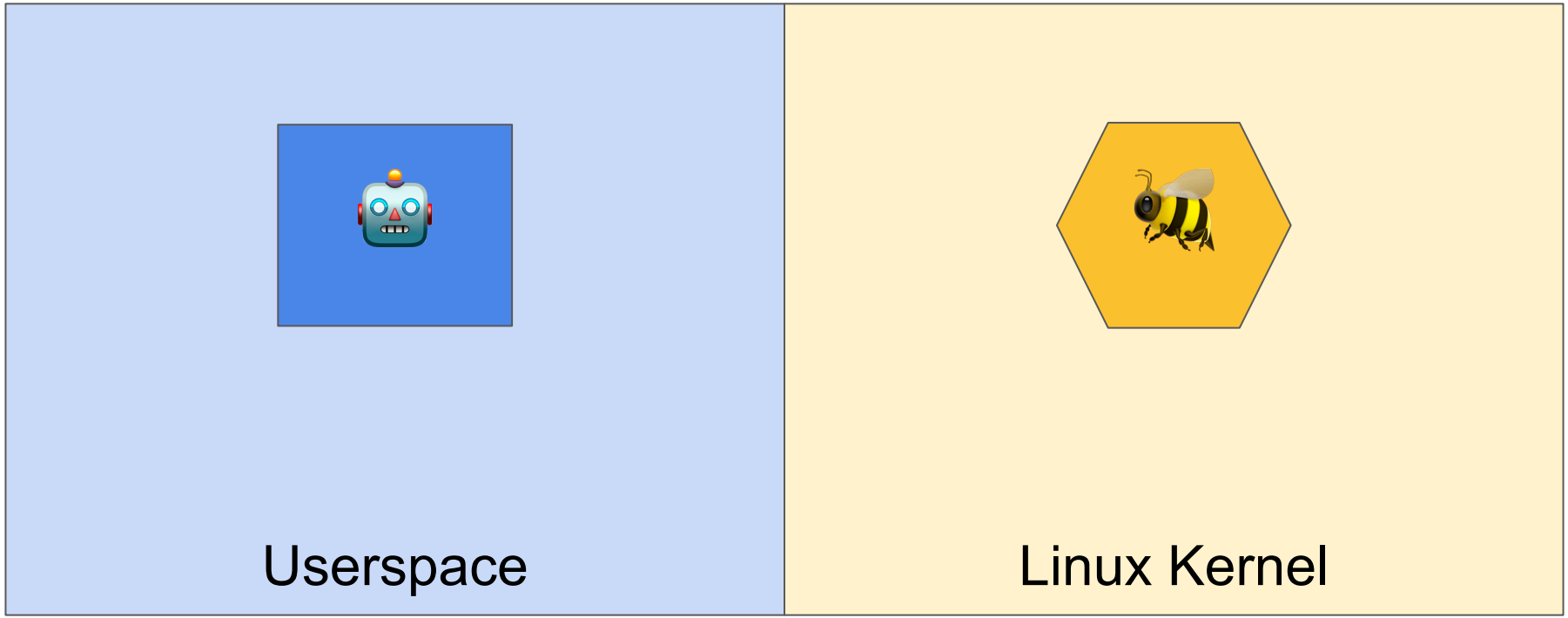
App v1: Fizz Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3.

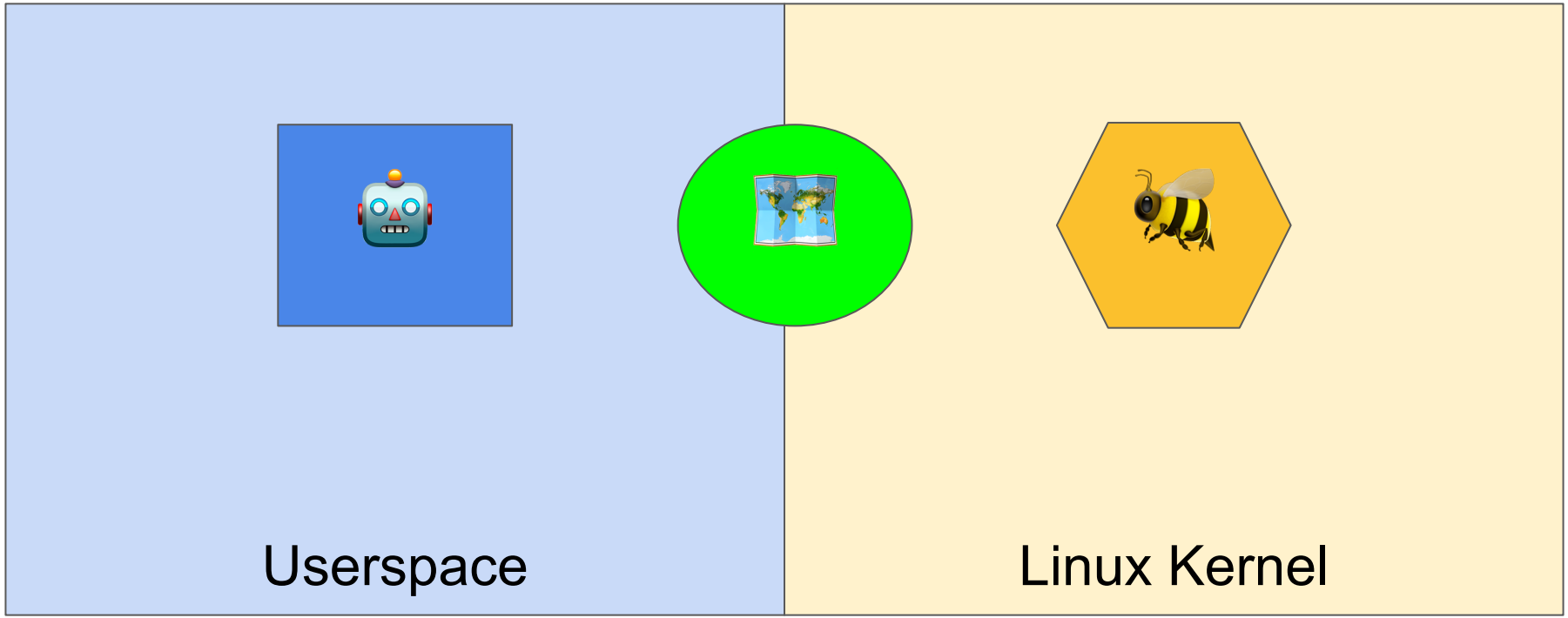
App v1: Fizz Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3.
Otherwise, just return XDP_PASS.

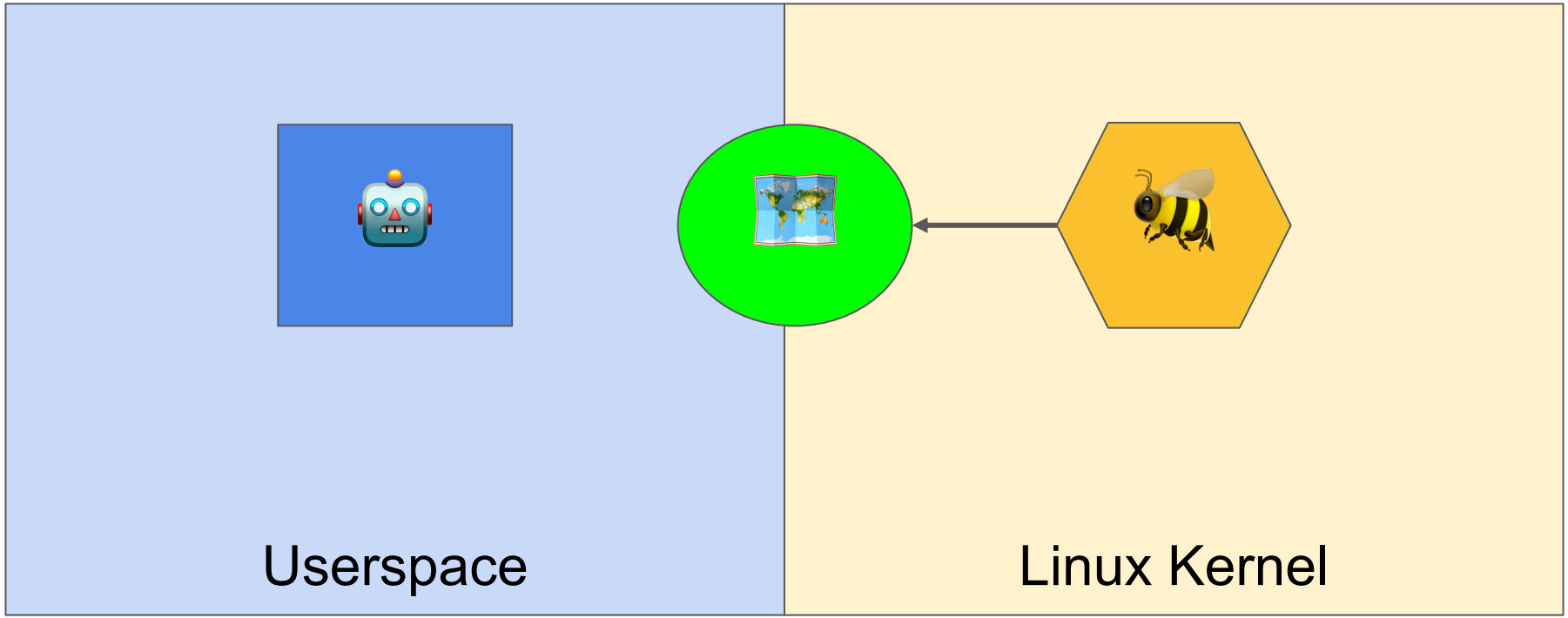
App v1: Fizz Feature



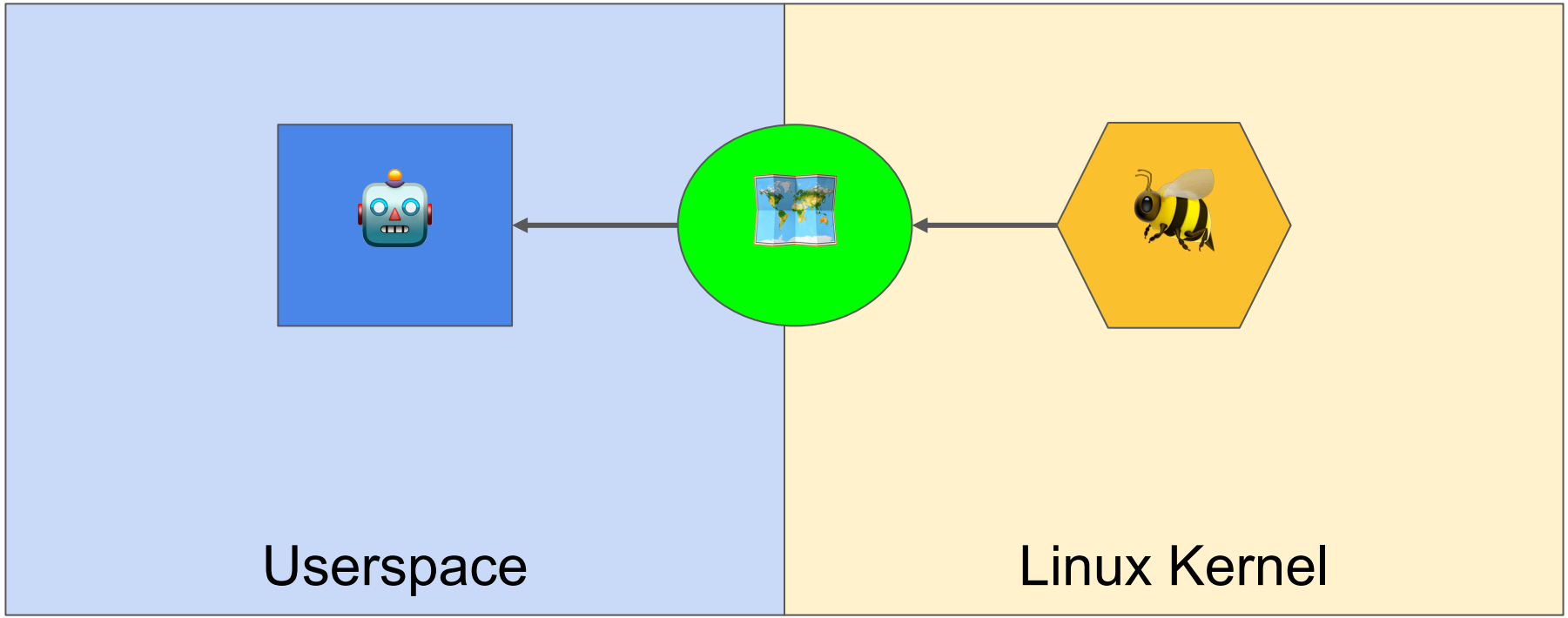
App v1: Fizz Feature



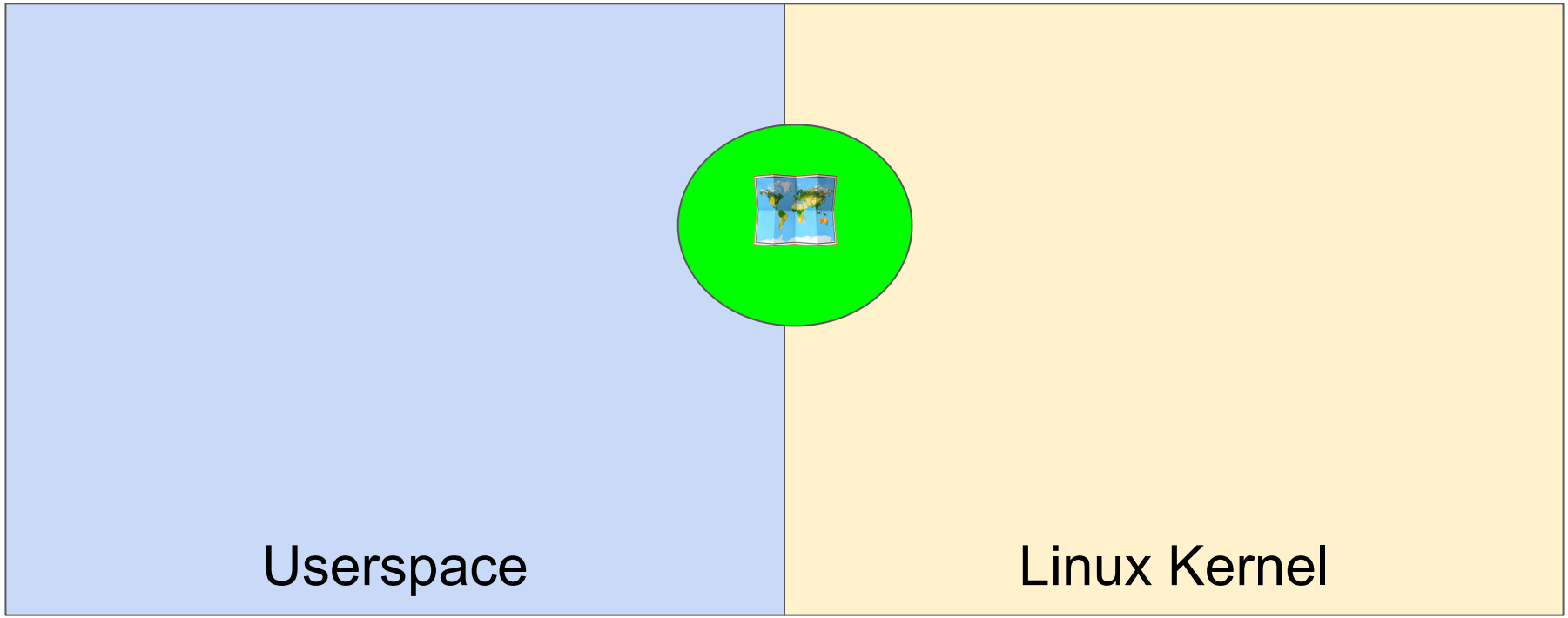
App v1: Fizz Feature



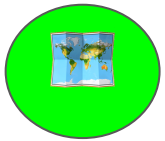
App v1: Fizz Feature



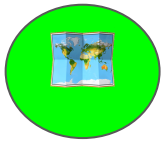
App v1: Fizz Feature



App v1: Fizz Feature

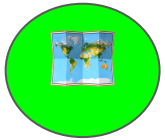


App v1: Fizz Feature



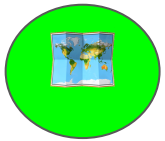
```rust
pub enum SourceAddr {
    Fizz,
}
```

App v1: Fizz Feature



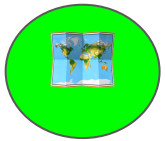
```rust
#[repr(C)]


pub enum SourceAddr {
    Fizz,
}
```

App v1: Fizz Feature



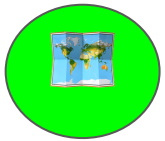
```rust
#[repr(C)]
#[derive(Clone, Copy)]

pub enum SourceAddr {
    Fizz,
}
```

App v1: Fizz Feature



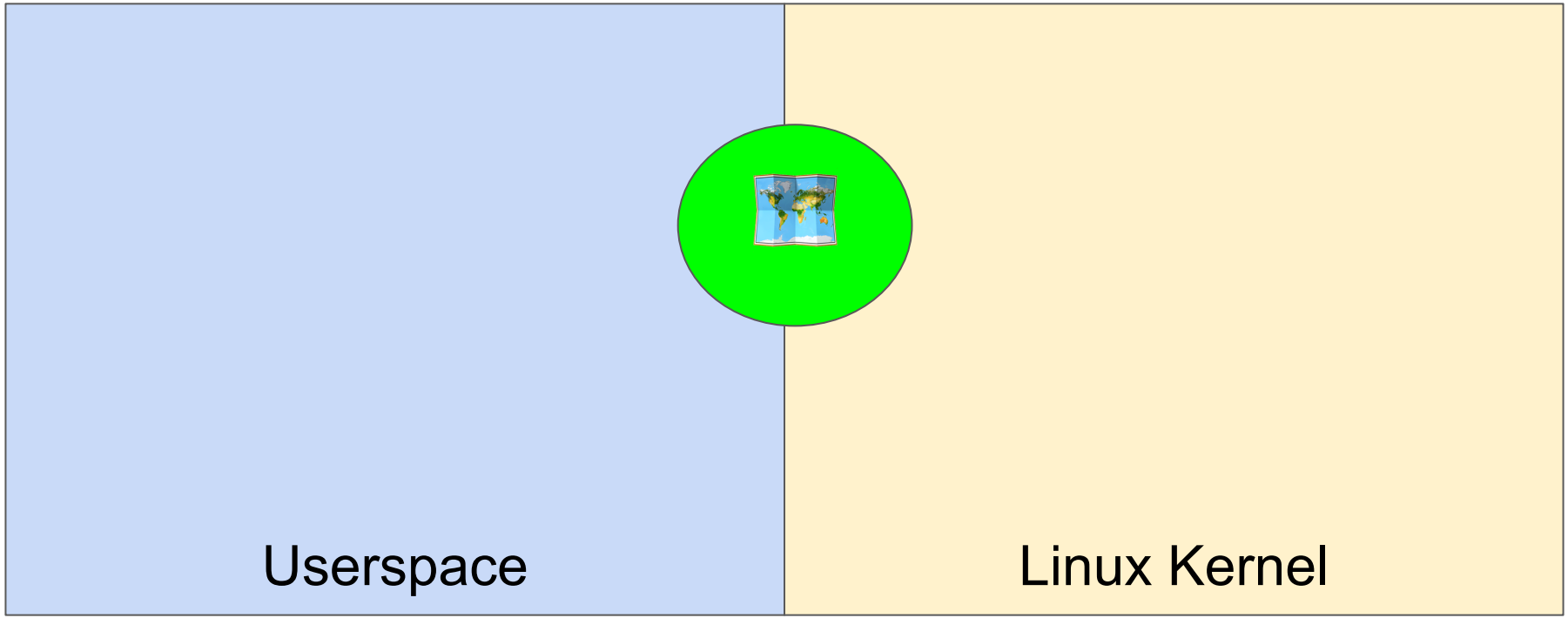
```rust
#[repr(C)]
#[derive(Clone, Copy)]

pub enum SourceAddr {
    Fizz,
}

#[cfg(feature = "user")]
unsafe impl aya::Pod for SourceAddr {}
```

App v1: Fizz Feature

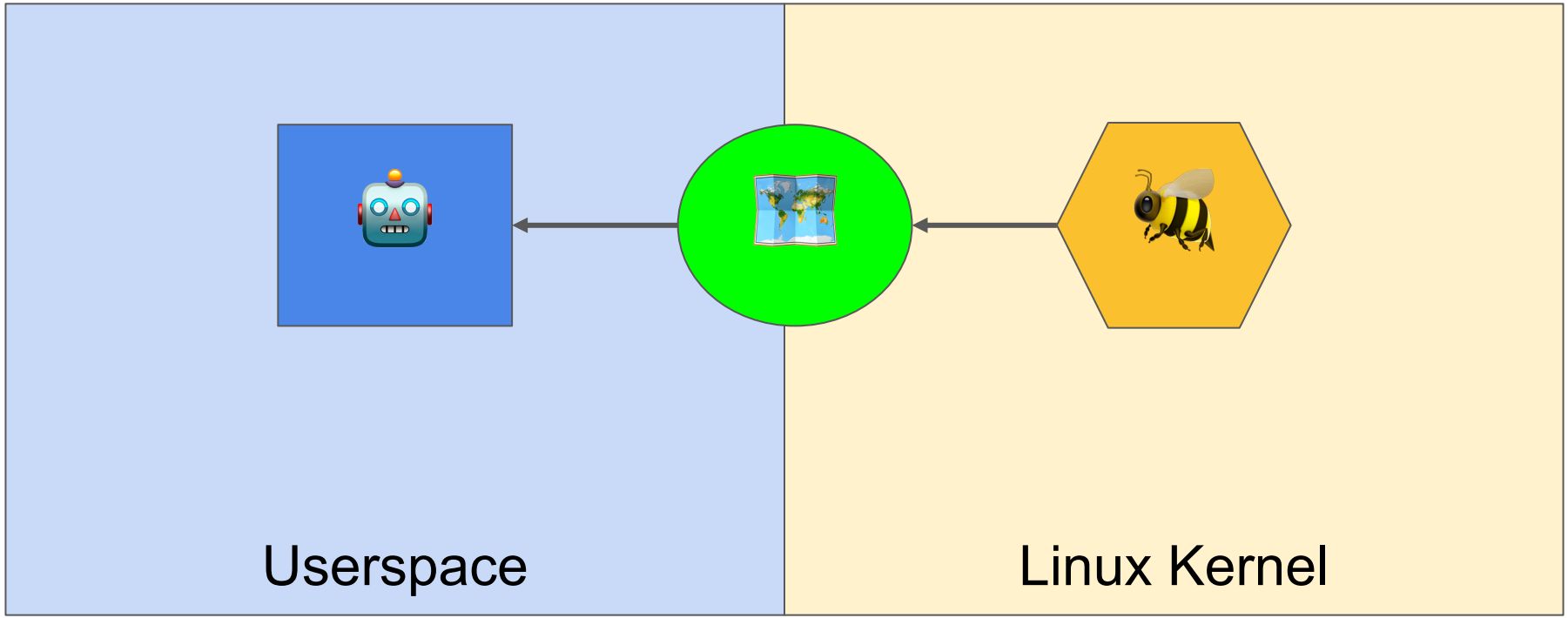


```rust
#[repr(C)]
#[derive(Clone, Copy)]
#[cfg_attr(feature = "user", derive(Debug))]
pub enum SourceAddr {
    Fizz,
}

#[cfg(feature = "user")]
unsafe impl aya::Pod for SourceAddr {}
```

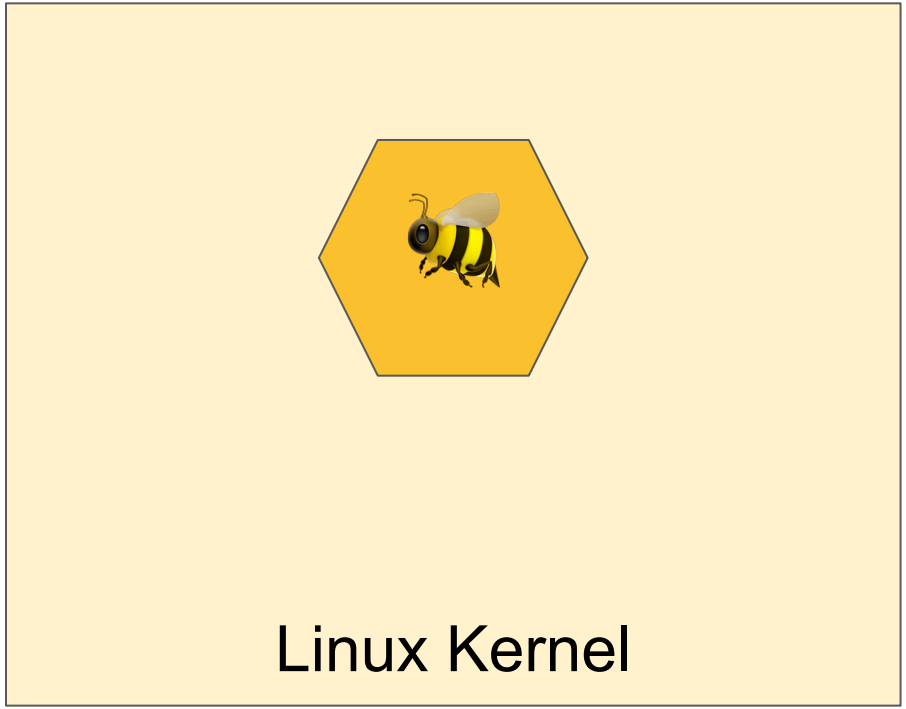
App v1: Fizz Feature



App v1: Fizz Feature



App v1: Fizz Feature



App v1: Fizz Feature

App v1: Fizz Feature

```rust
#[map]
pub static mut SOURCE_ADDR_QUEUE: Queue<SourceAddr> = Queue::with_max_entries(1024, 0);
```

App v1: Fizz Feature

```
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {









}
```

App v1: Fizz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
}
```

App v1: Fizz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    info!(ctx, "IPv4 Source Address: {}", source_addr);









}
```

App v1: Fizz Feature

```
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    info!(ctx, "IPv4 Source Address: {}", source_addr);









}
```

App v1: Fizz Feature

```
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    info!(ctx, "IPv4 Source Address: {}", source_addr);
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);









}
```

Bencher

App v1: Fizz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    info!(ctx, "IPv4 Source Address: {}", source_addr);
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);

    if let Some(source_addr) = opt_source_addr {




    }


}
```

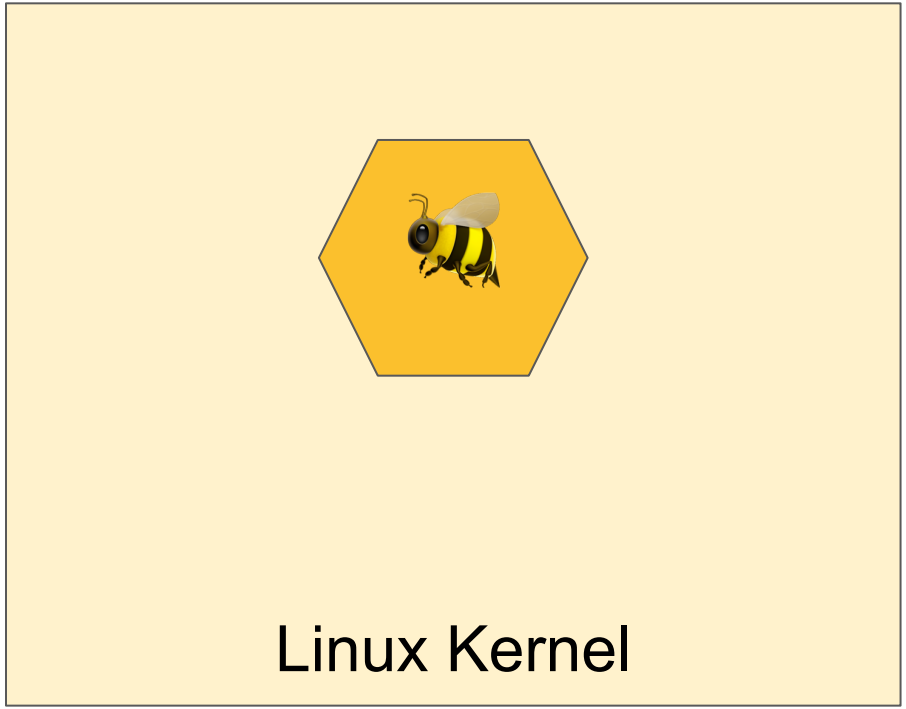
App v1: Fizz Feature

```
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    info!(ctx, "IPv4 Source Address: {}", source_addr);
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);

    if let Some(source_addr) = opt_source_addr {
        unsafe {
            if let Err(e) = SOURCE_ADDR_QUEUE.push(&source_addr, 0) {
                error!(ctx, "Failed to push source address into queue: {}", e);
            }
        }
    }


}
```

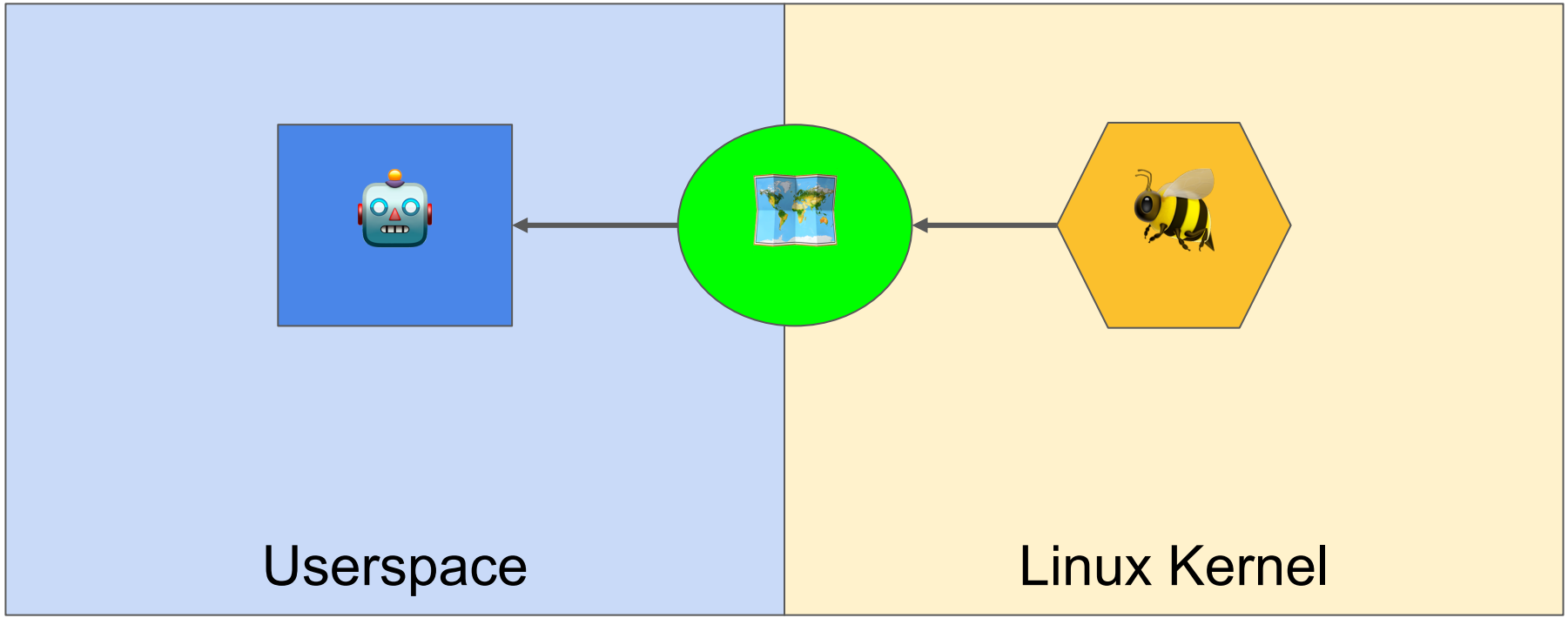
App v1: Fizz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    info!(ctx, "IPv4 Source Address: {}", source_addr);
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);

    if let Some(source_addr) = opt_source_addr {
        unsafe {
            if let Err(e) = SOURCE_ADDR_QUEUE.push(&source_addr, 0) {
                error!(ctx, "Failed to push source address into queue: {}", e);
            }
        }
    }

    Ok(xdp_action::XDP_PASS)
}
```

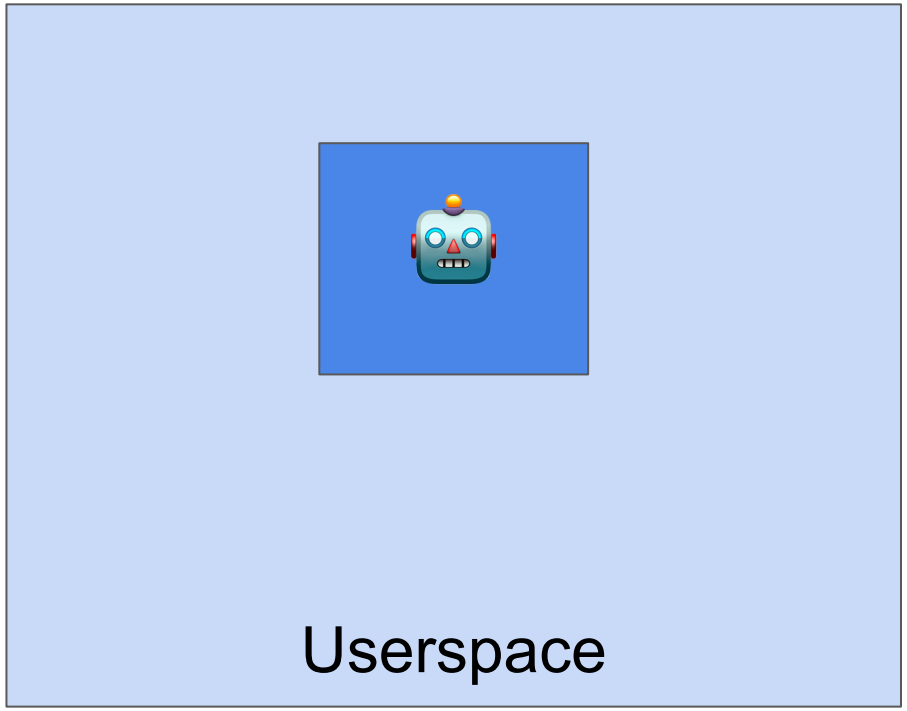
App v1: Fizz Feature



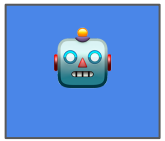
App v1: Fizz Feature



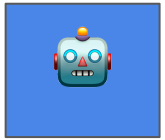
App v1: Fizz Feature



App v1: Fizz Feature

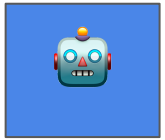


App v1: Fizz Feature



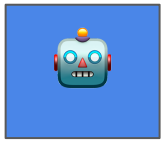
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    ````
}
```
```

App v1: Fizz Feature



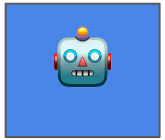
```
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    ...

    ```
}
```

Bencher
```

App v1: Fizz Feature



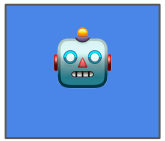
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    ...

    spawn_agent(&mut bpf).await?;

    ```
}
```
```

App v1: Fizz Feature



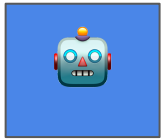
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    ...

    spawn_agent(&mut bpf).await?;

    info!("Waiting for Ctrl-C...");
    signal::ctrl_c().await?;

    ```
}
```
```

App v1: Fizz Feature



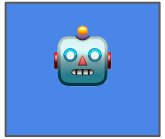
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    ...

    spawn_agent(&mut bpf).await?;

    info!("Waiting for Ctrl-C...");
    signal::ctrl_c().await?;
    info!("Exiting...");
    Ok(())
}
```

App v1: Fizz Feature



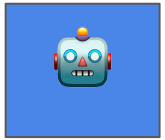
```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {









}
```

App v1: Fizz Feature



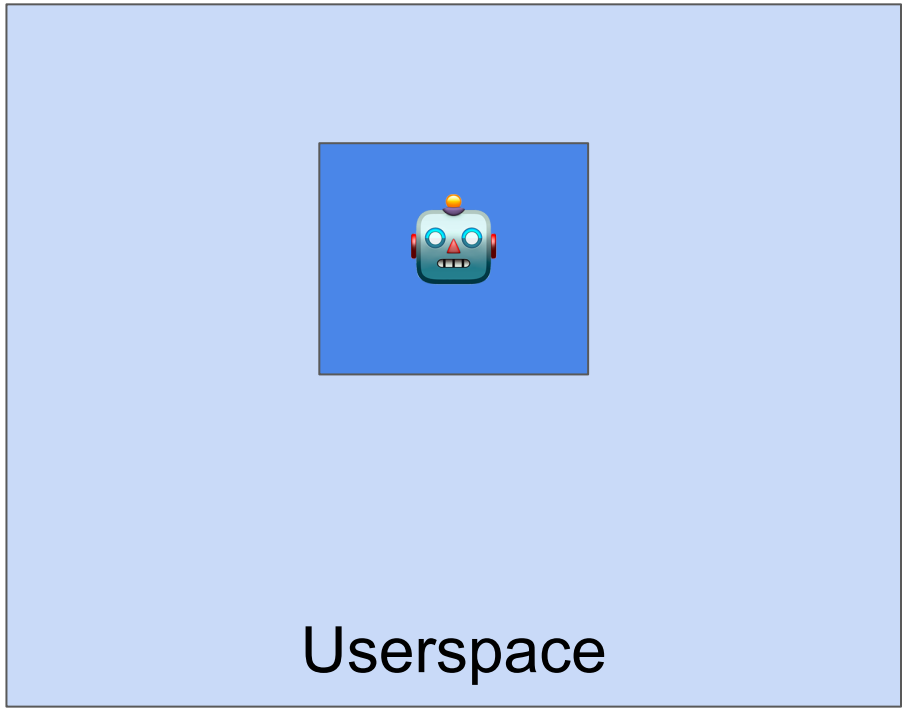
```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;









}
```

App v1: Fizz Feature

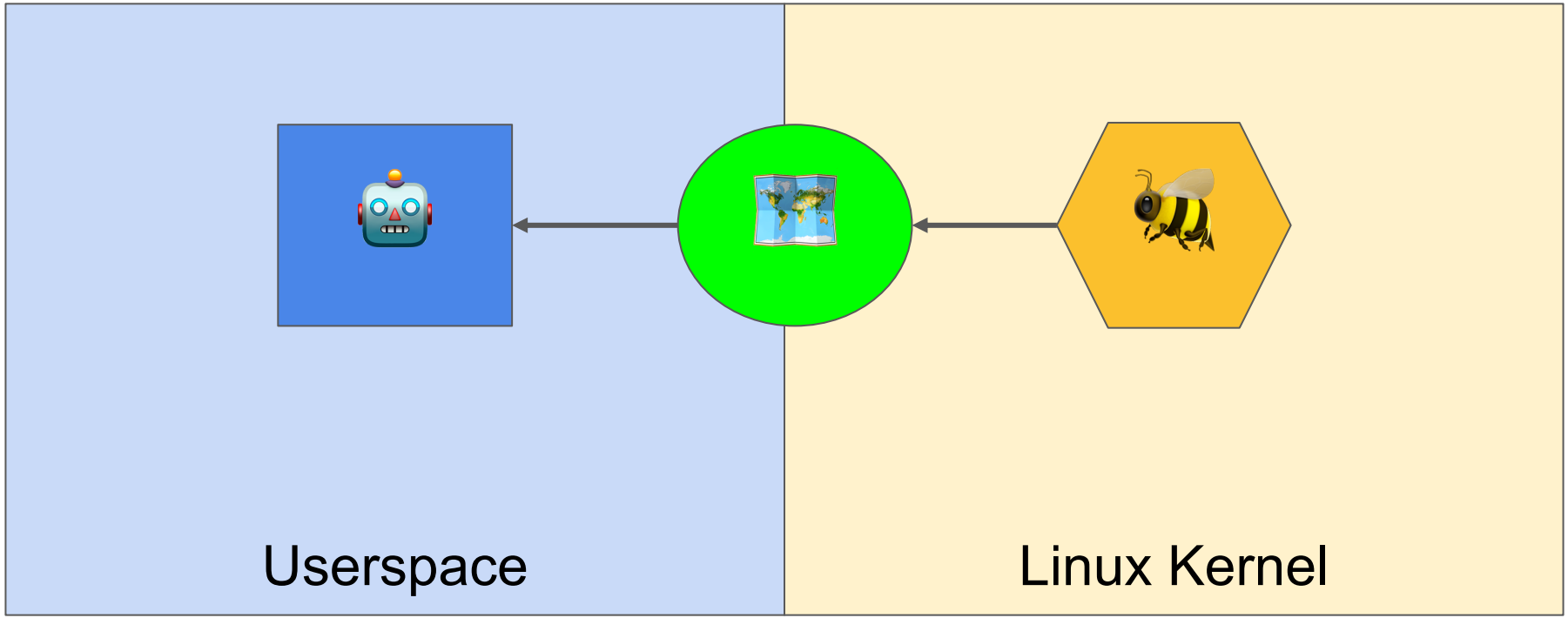


```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;

    loop {
        while let Ok(source_addr) = xdp_map.pop(0) {
            info!("{:?}", source_addr);
        }
    }
}
```

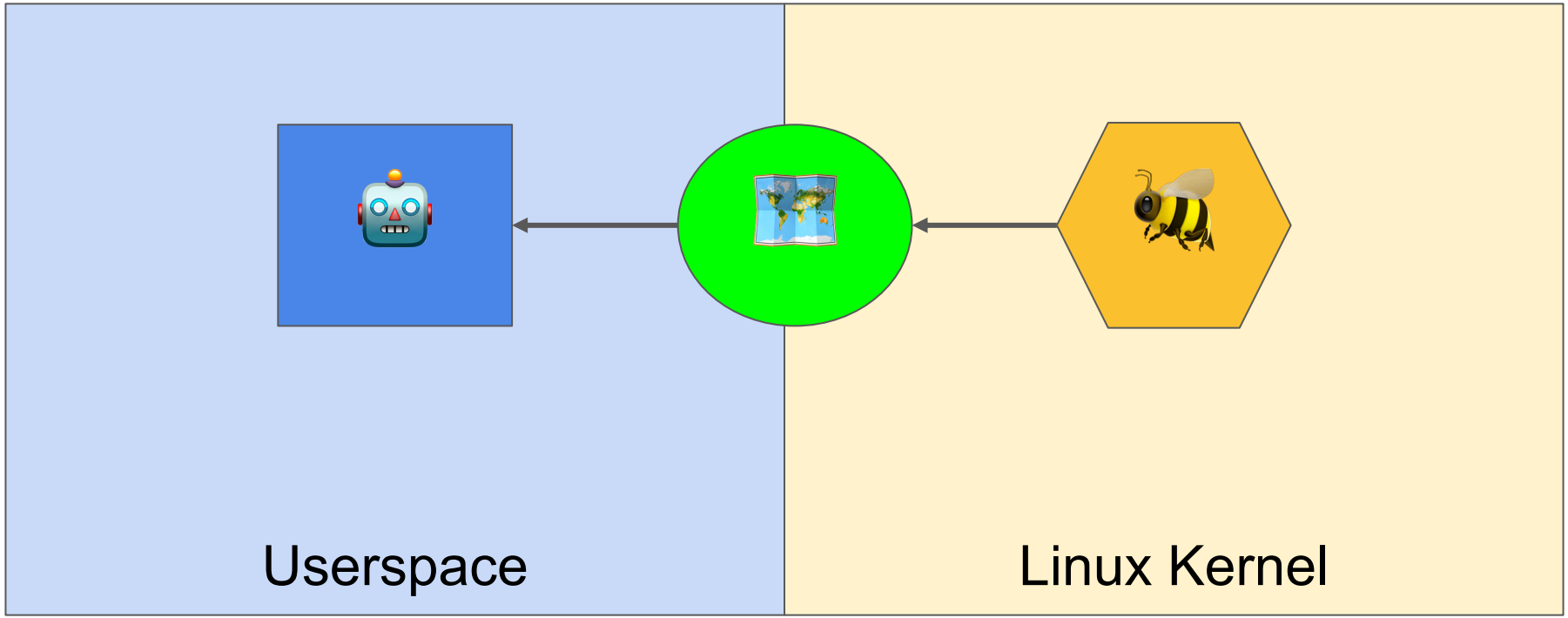
App v1: Fizz Feature



App v1: Fizz Feature



App v1: Fizz Feature



Add a simple update

App v2: FizzBuzz Feature

App v2: FizzBuzz Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3,

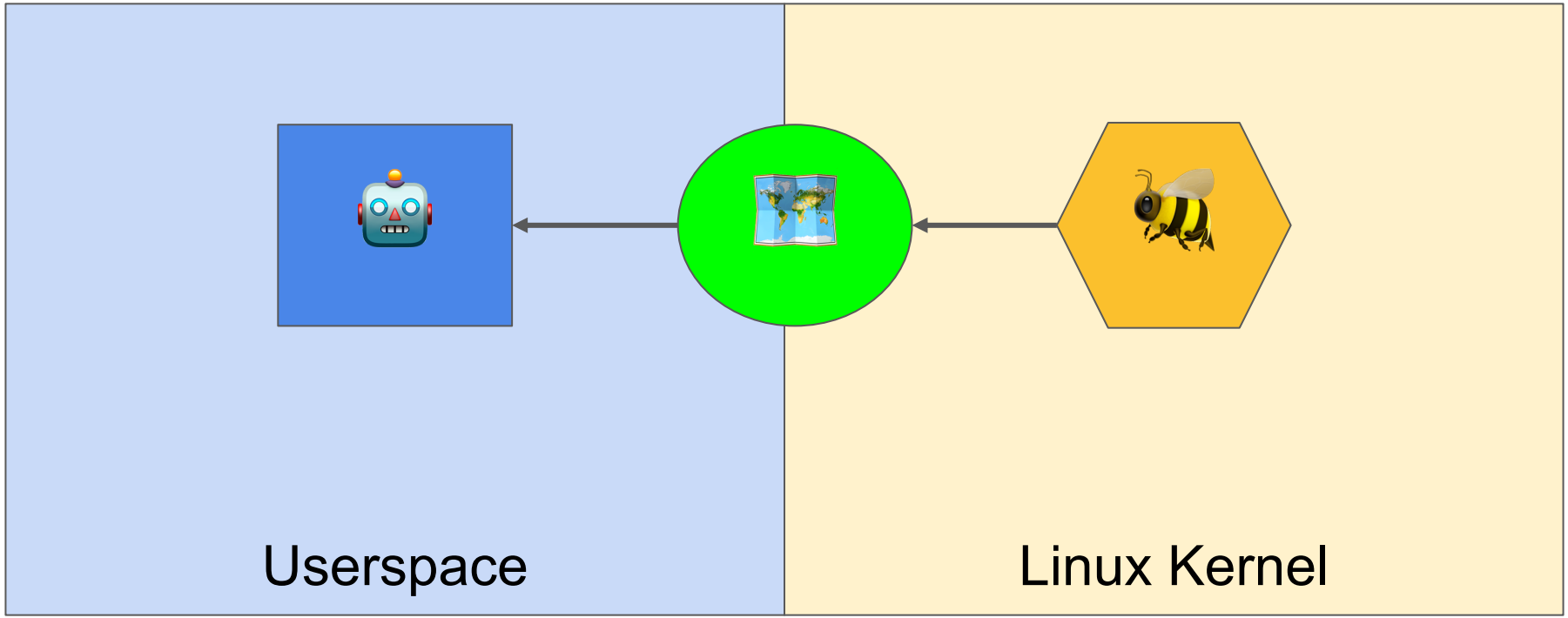
App v2: FizzBuzz Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3,
“Buzz” if divisible by 5, or “FizzBuzz” if both.

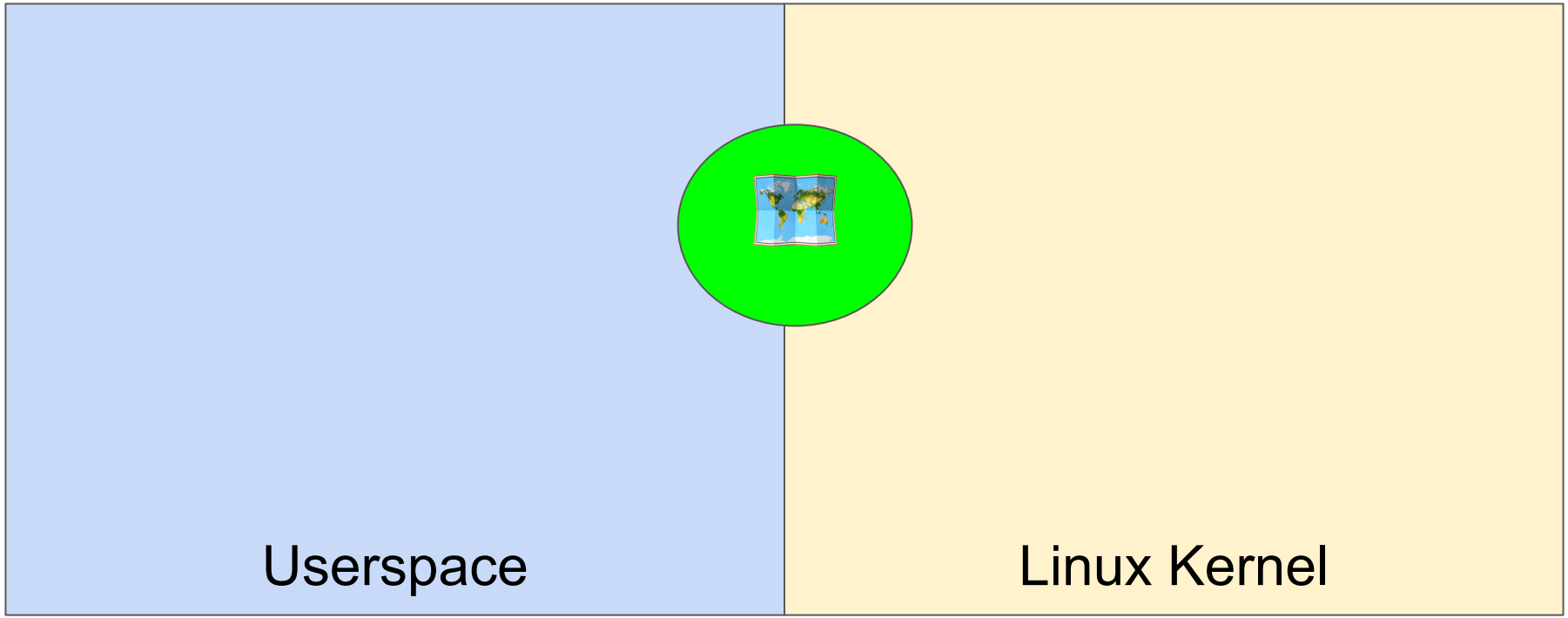
App v2: FizzBuzz Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3,
“Buzz” if divisible by 5, or “FizzBuzz” if both.
Otherwise, just return XDP_PASS.

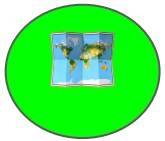
App v2: FizzBuzz Feature



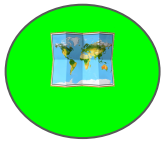
App v2: FizzBuzz Feature



App v2: FizzBuzz Feature

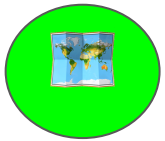


App v2: FizzBuzz Feature



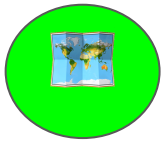
```
#[repr(C)]
#[derive(Clone, Copy)]
#[cfg_attr(feature = "user", derive(Debug))]
pub enum SourceAddr {
    Fizz,


}

#[cfg(feature = "user")]
unsafe impl aya::Pod for SourceAddr {}
```

App v2: FizzBuzz Feature



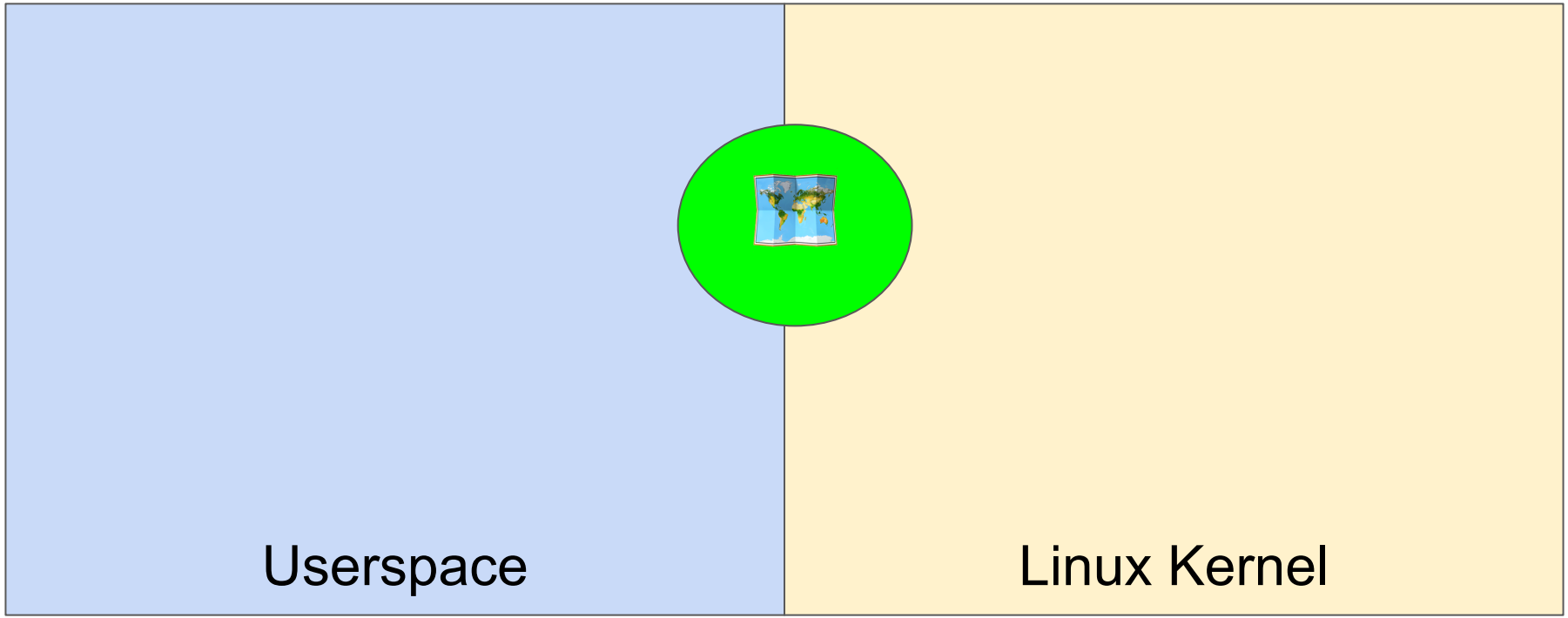
```rust
#[repr(C)]
#[derive(Clone, Copy)]
#[cfg_attr(feature = "user", derive(Debug))]
pub enum SourceAddr {
    Fizz,
    Buzz,

}

#[cfg(feature = "user")]
unsafe impl aya::Pod for SourceAddr {}
```

App v2: FizzBuzz Feature

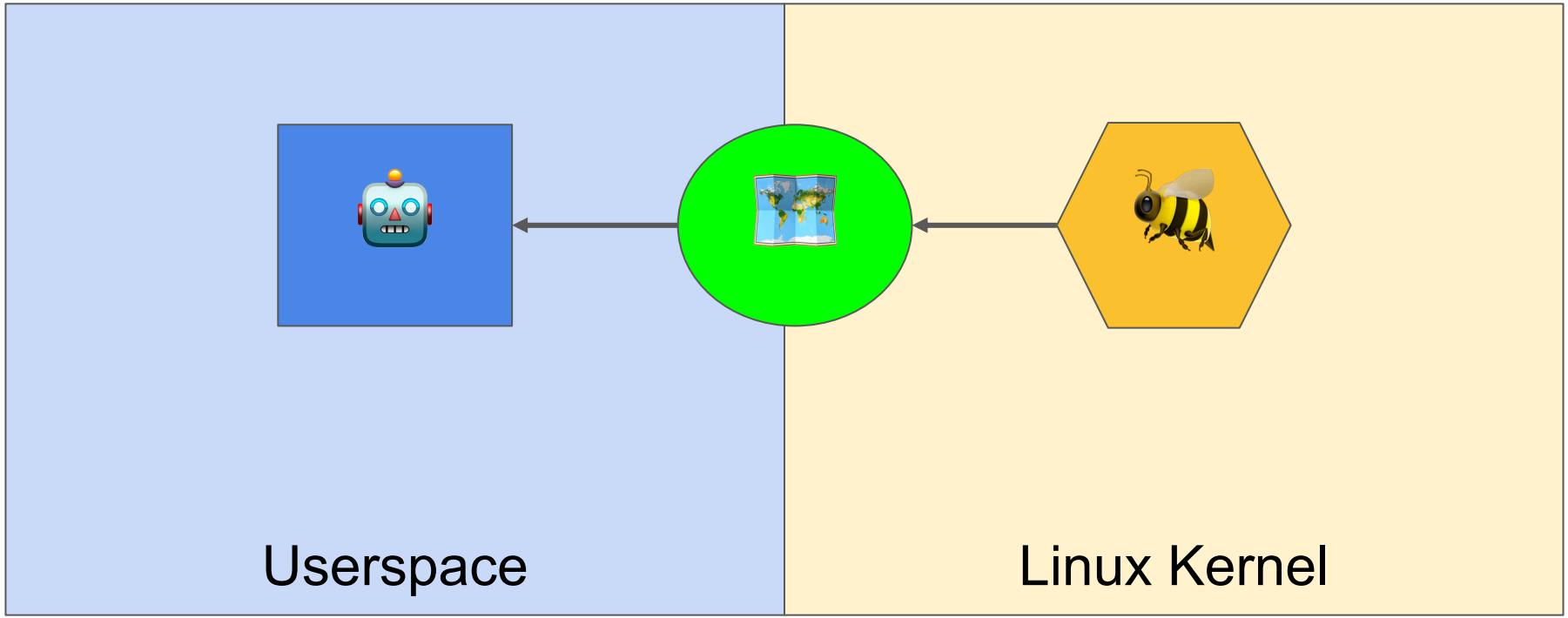


```rust
#[repr(C)]
#[derive(Clone, Copy)]
#[cfg_attr(feature = "user", derive(Debug))]
pub enum SourceAddr {
    Fizz,
    Buzz,
    FizzBuzz,
}

#[cfg(feature = "user")]
unsafe impl aya::Pod for SourceAddr {}
```

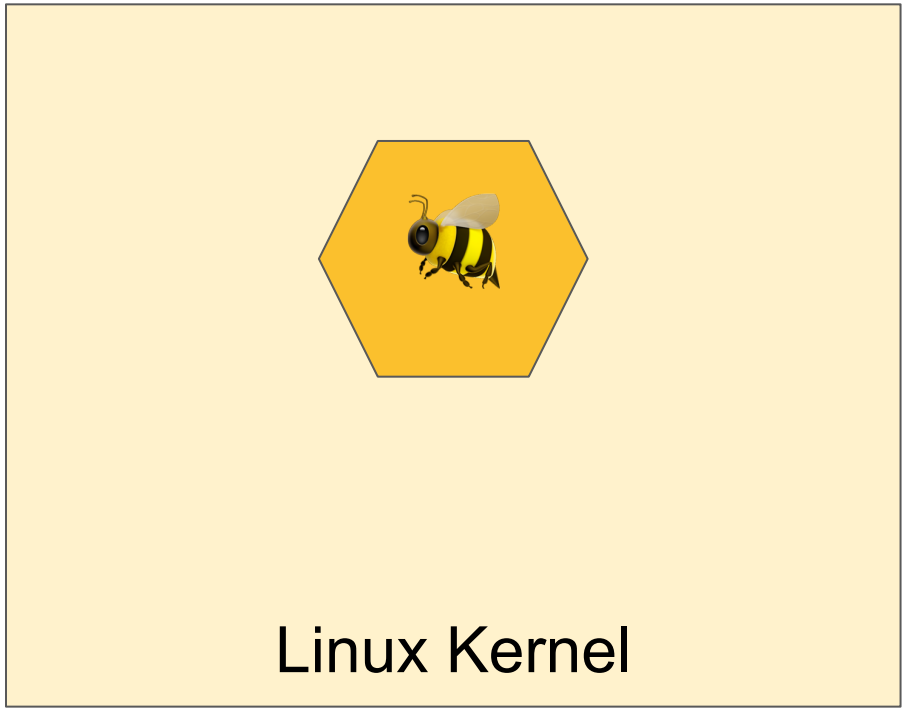
App v2: FizzBuzz Feature



App v2: FizzBuzz Feature



App v2: FizzBuzz Feature



App v2: FizzBuzz Feature

App v2: FizzBuzz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {

}
```

App v2: FizzBuzz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...









}
```

App v2: FizzBuzz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);









}
```

App v2: FizzBuzz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);









}
```

App v2: FizzBuzz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);
    let opt_source_addr = match (source_addr % 3, source_addr % 5) {
        (0, 0) => Some(SourceAddr::FizzBuzz),
        (0, _) => Some(SourceAddr::Fizz),
        (_, 0) => Some(SourceAddr::Buzz),
        _ => None,
    };




}
```

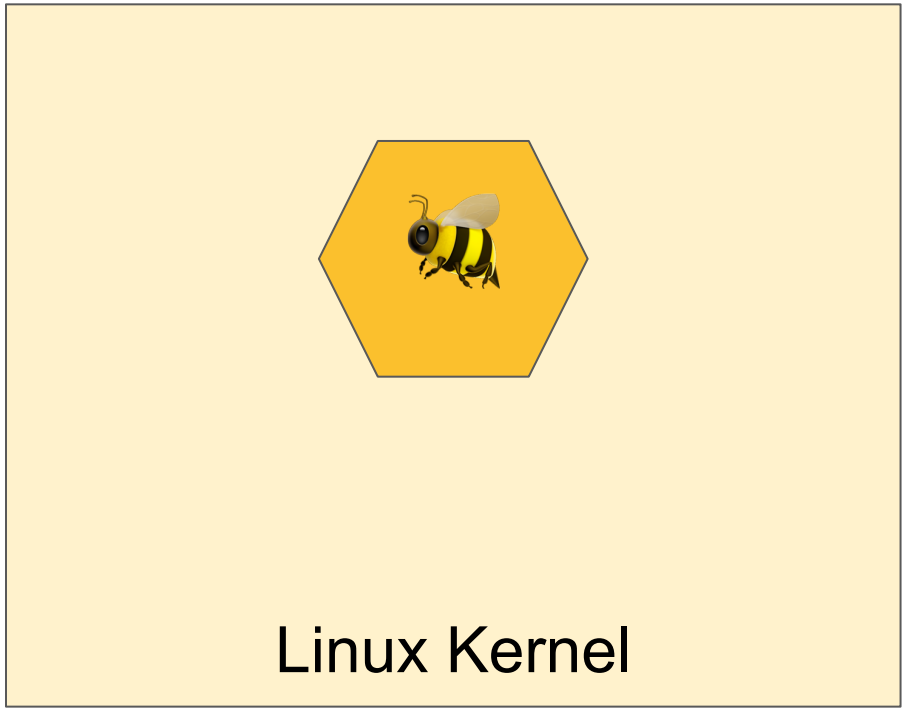
App v2: FizzBuzz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);
    let opt_source_addr = match (source_addr % 3, source_addr % 5) {
        (0, 0) => Some(SourceAddr::FizzBuzz),
        (0, _) => Some(SourceAddr::Fizz),
        (_, 0) => Some(SourceAddr::Buzz),
        _ => None,
    };

    ...

}
```

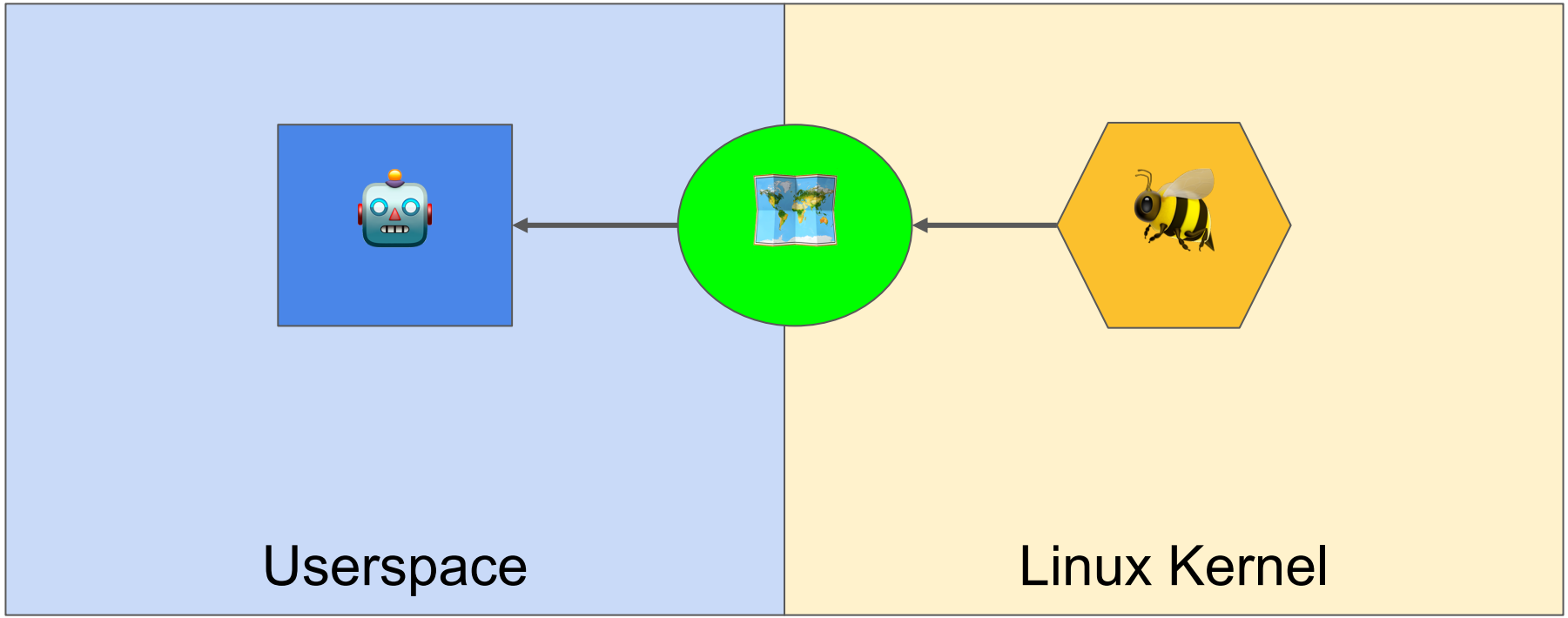
App v2: FizzBuzz Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = (source_addr % 3 == 0).then_some(SourceAddr::Fizz);
    let opt_source_addr = match (source_addr % 3, source_addr % 5) {
        (0, 0) => Some(SourceAddr::FizzBuzz),
        (0, _) => Some(SourceAddr::Fizz),
        (_, 0) => Some(SourceAddr::Buzz),
        _ => None,
    };

    ...
    Ok(xdp_action::XDP_PASS)
}
```

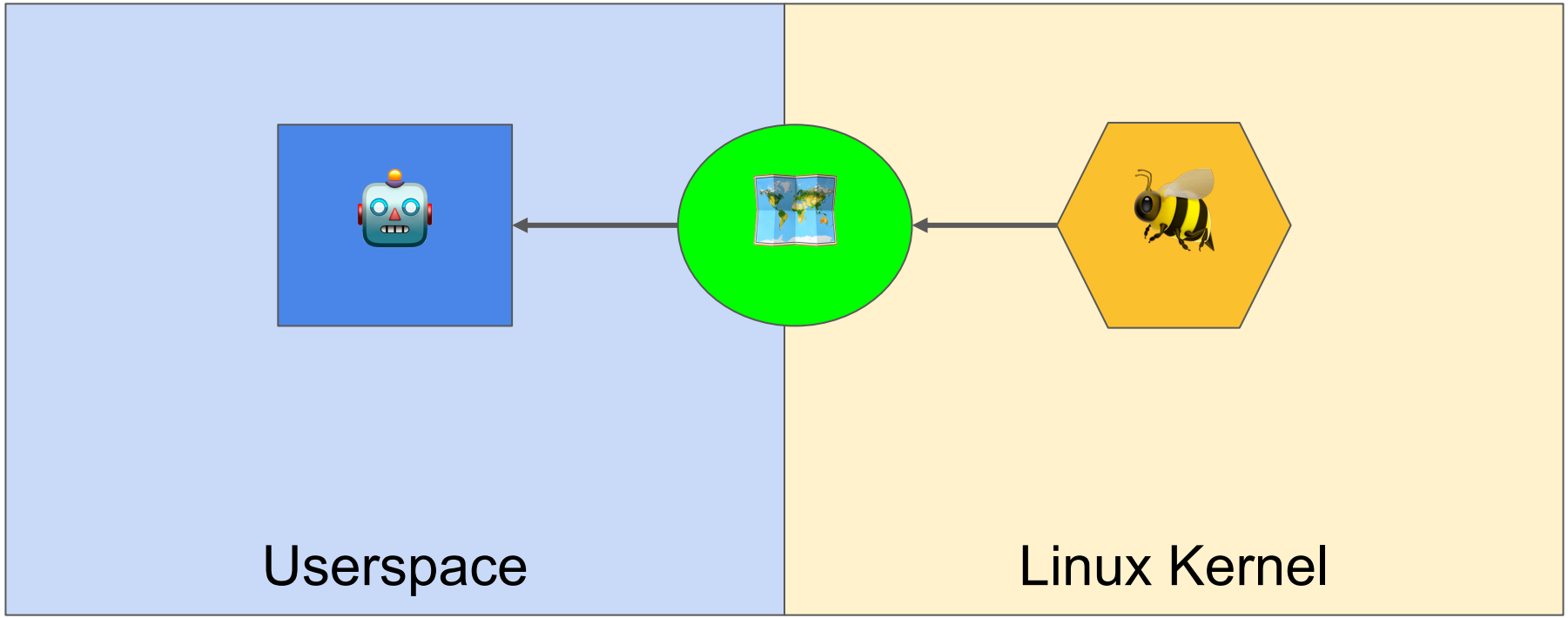
App v2: FizzBuzz Feature



App v2: FizzBuzz Feature



App v2: FizzBuzz Feature



Add another (seemingly) simple update

App v3: FizzBuzzFibonacci Feature

App v3: FizzBuzzFibonacci Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3,

App v3: FizzBuzzFibonacci Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3,
“Buzz” if divisible by 5, or “FizzBuzz” if both.

App v3: FizzBuzzFibonacci Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3,
“Buzz” if divisible by 5, or “FizzBuzz” if both.

Except if the remainder of the IPv4 source address divided by 256
is part of the fibonacci sequence, then push “Fibonacci”.

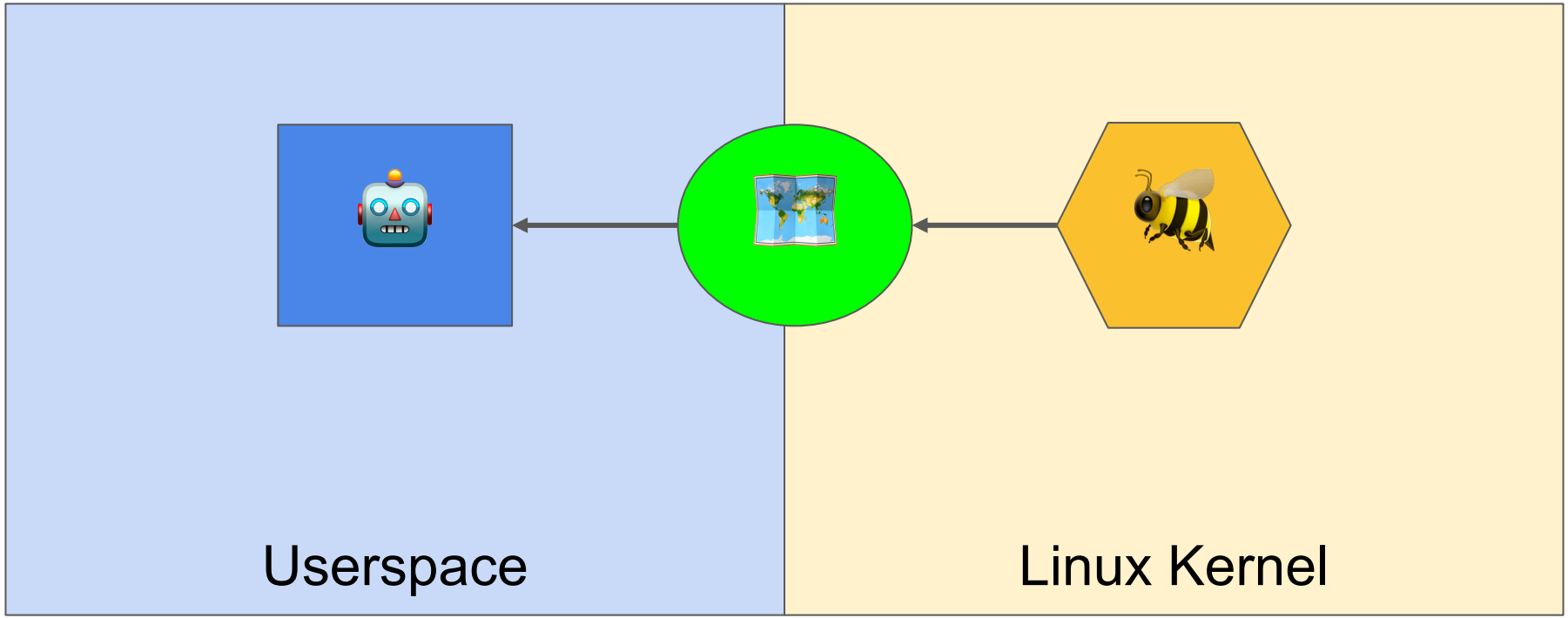
App v3: FizzBuzzFibonacci Feature

Push “Fizz” into the queue if IPv4 source address is divisible by 3,
“Buzz” if divisible by 5, or “FizzBuzz” if both.

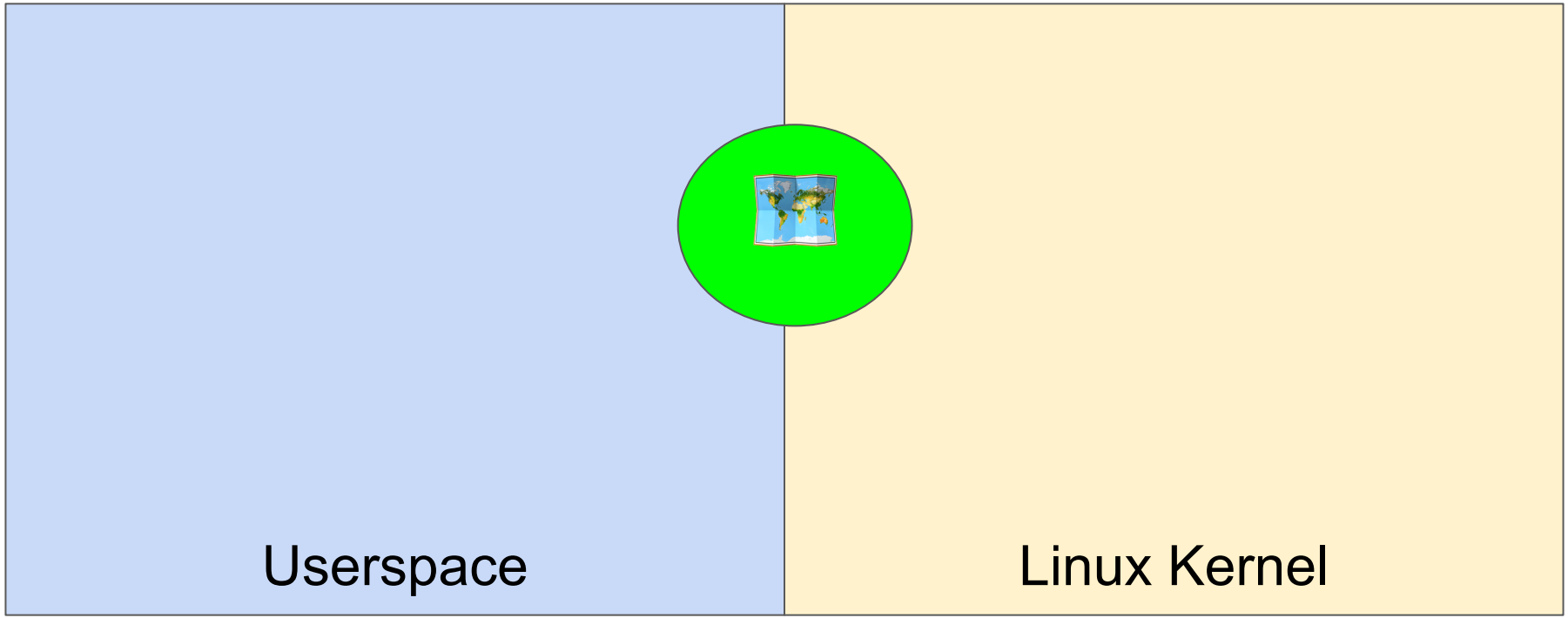
Except if the remainder of the IPv4 source address divided by 256
is part of the fibonacci sequence, then push “Fibonacci”.

Otherwise, just return XDP_PASS.

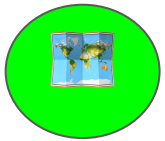
App v3: FizzBuzzFibonacci Feature



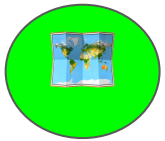
App v3: FizzBuzzFibonacci Feature



App v3: FizzBuzzFibonacci Feature

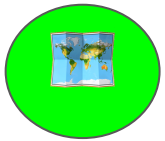


App v3: FizzBuzzFibonacci Feature



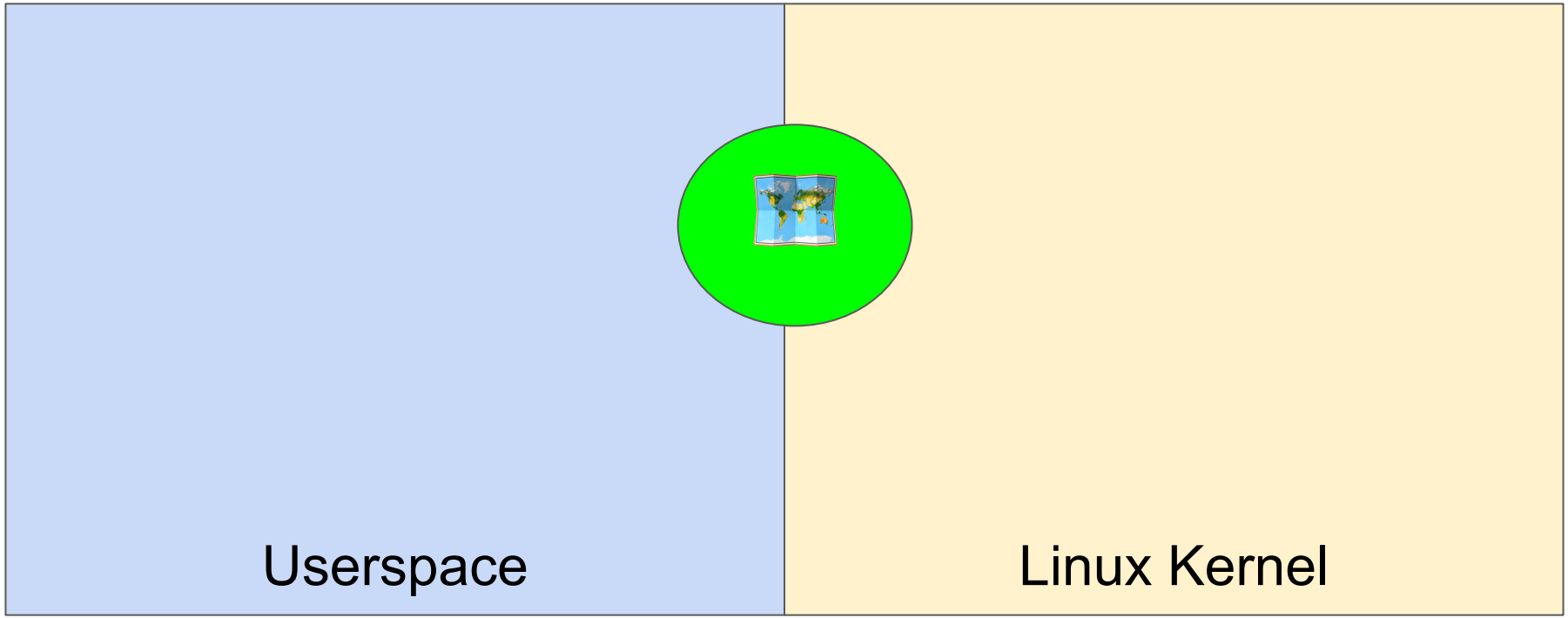
```rust
#[repr(C)]
#[derive(Clone, Copy)]
#[cfg_attr(feature = "user", derive(Debug))]
pub enum SourceAddr {
    Fizz,
    Buzz,
    FizzBuzz,

}

#[cfg(feature = "user")]
unsafe impl aya::Pod for SourceAddr {}
```

App v3: FizzBuzzFibonacci Feature

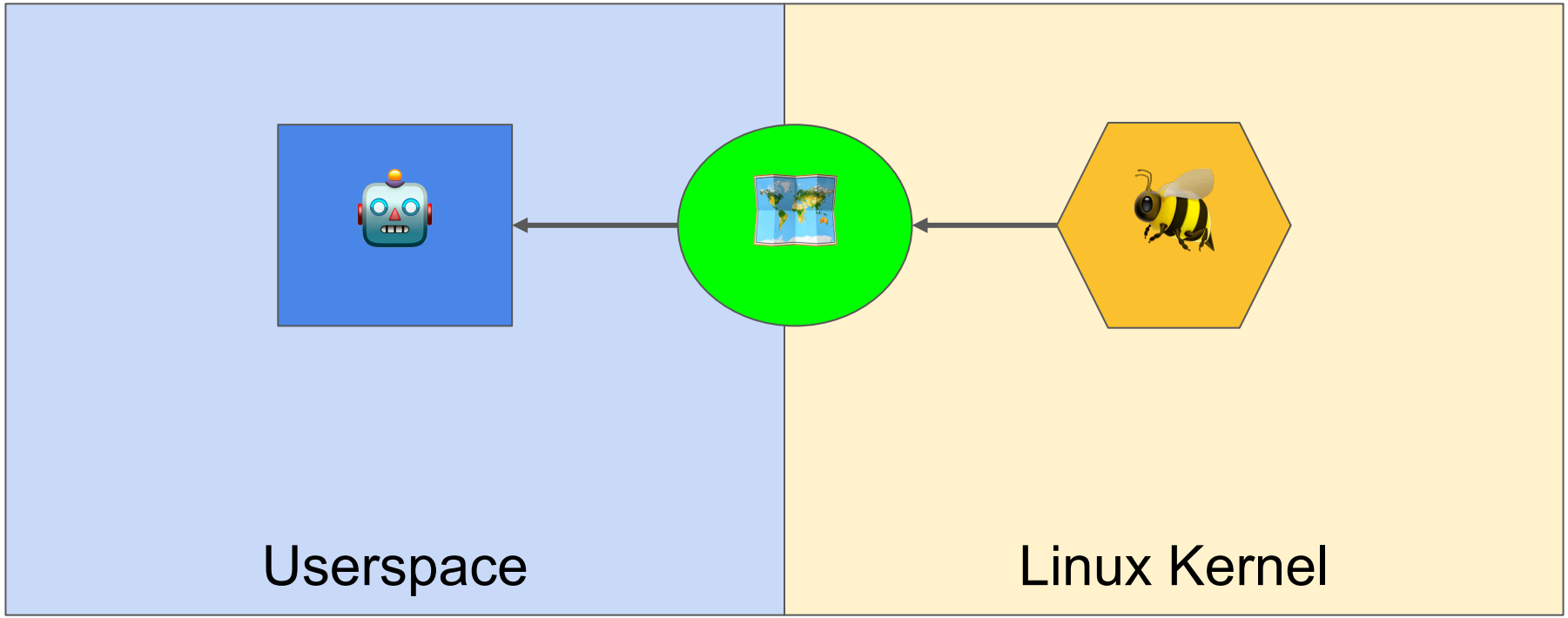


```rust
#[repr(C)]
#[derive(Clone, Copy)]
#[cfg_attr(feature = "user", derive(Debug))]
pub enum SourceAddr {
    Fizz,
    Buzz,
    FizzBuzz,
    Fibonacci,
}

#[cfg(feature = "user")]
unsafe impl aya::Pod for SourceAddr {}
```

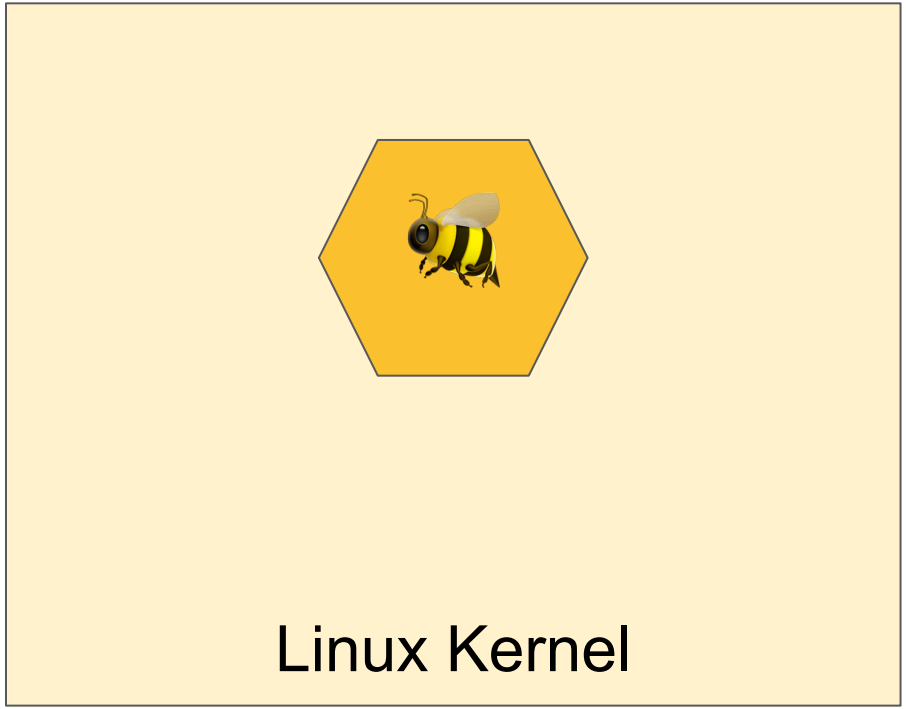
App v3: FizzBuzzFibonacci Feature



App v3: FizzBuzzFibonacci Feature



App v3: FizzBuzzFibonacci Feature



App v3: FizzBuzzFibonacci Feature

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {

}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = match (source_addr % 3, source_addr % 5) { ... };









}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = match (source_addr % 3, source_addr % 5) { ... };









}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = match (source_addr % 3, source_addr % 5) { ... };
    let opt_source_addr = is_fibonacci(source_addr as u8)
        .then_some(SourceAddr::Fibonacci)
        .or(match (source_addr % 3, source_addr % 5) {
            (0, 0) => Some(SourceAddr::FizzBuzz),
            (0, _) => Some(SourceAddr::Fizz),
            (_, 0) => Some(SourceAddr::Buzz),
            _ => None,
        });



}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = match (source_addr % 3, source_addr % 5) { ... };
    let opt_source_addr = is_fibonacci(source_addr as u8)
        .then_some(SourceAddr::Fibonacci)
        .or(match (source_addr % 3, source_addr % 5) {
            (0, 0) => Some(SourceAddr::FizzBuzz),
            (0, _) => Some(SourceAddr::Fizz),
            (_, 0) => Some(SourceAddr::Buzz),
            _ => None,
        });

    ...

}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = match (source_addr % 3, source_addr % 5) { ... };
    let opt_source_addr = is_fibonacci(source_addr as u8)
        .then_some(SourceAddr::Fibonacci)
        .or(match (source_addr % 3, source_addr % 5) {
            (0, 0) => Some(SourceAddr::FizzBuzz),
            (0, _) => Some(SourceAddr::Fizz),
            (_, 0) => Some(SourceAddr::Buzz),
            _ => None,
        });

    ...
    Ok(xdp_action::XDP_PASS)
}
```

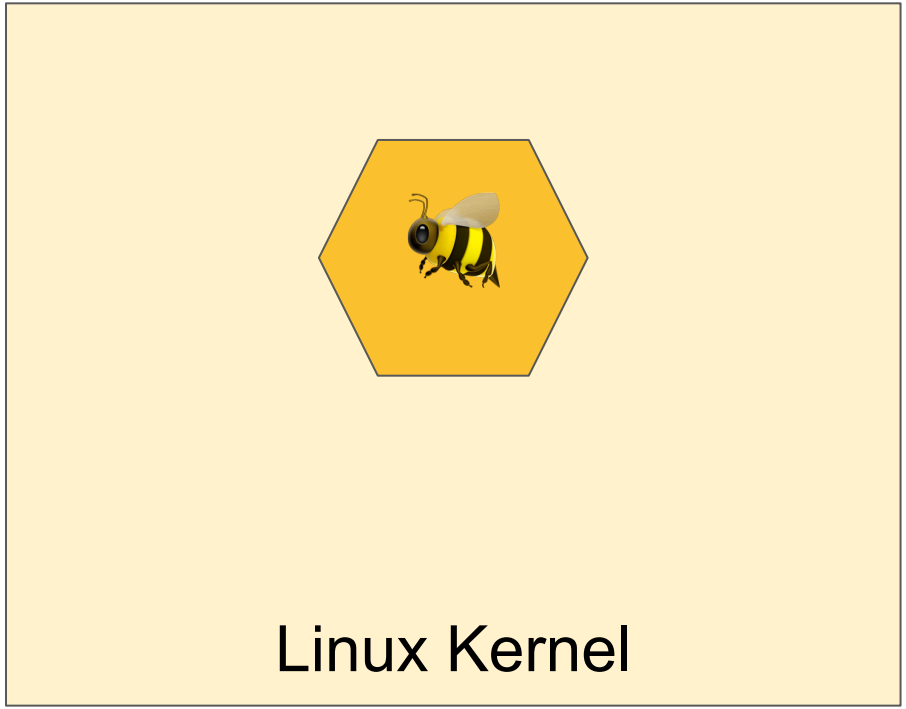
App v3: FizzBuzzFibonacci Feature

```rust
fn is_fibonacci(n: u8) -> bool {









}
```

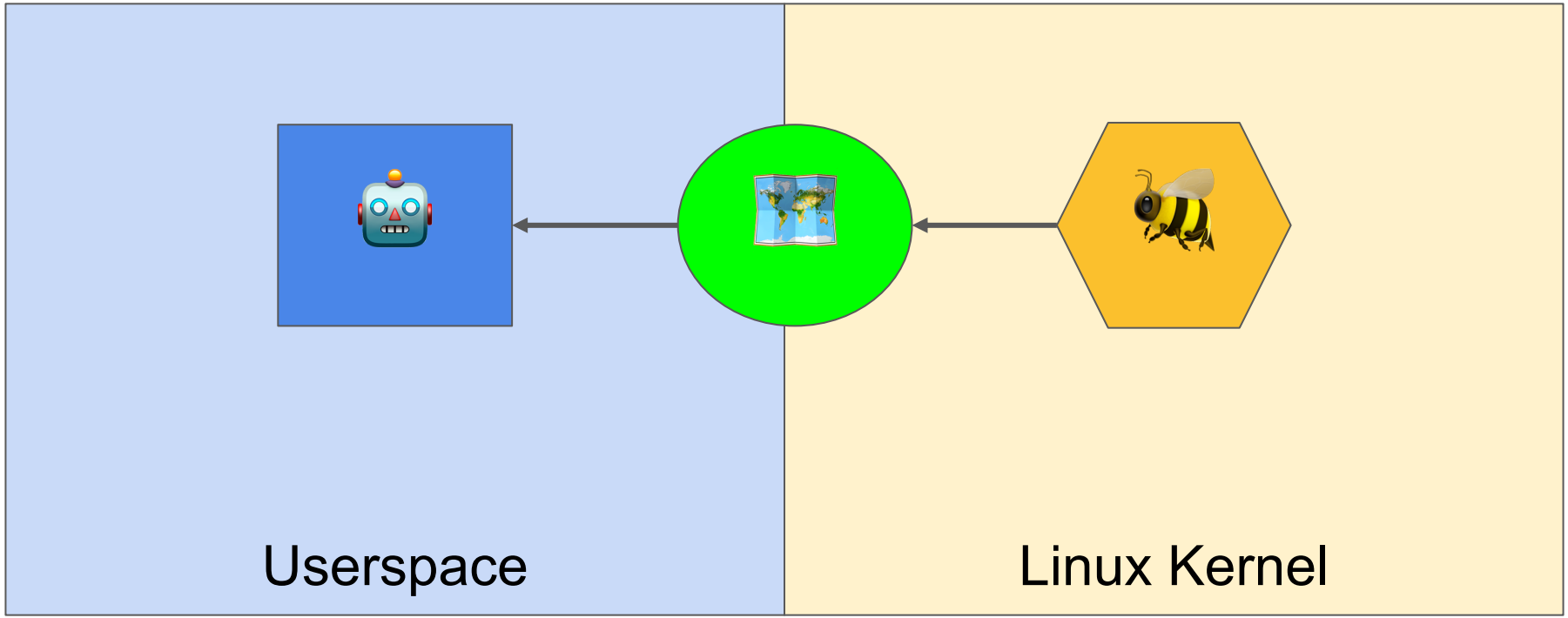
App v3: FizzBuzzFibonacci Feature

```rust
fn is_fibonacci(n: u8) -> bool {
    let (mut a, mut b) = (0, 1);
    while b < n {
        let c = a + b;
        a = b;
        b = c;
    }
    b == n
}
```

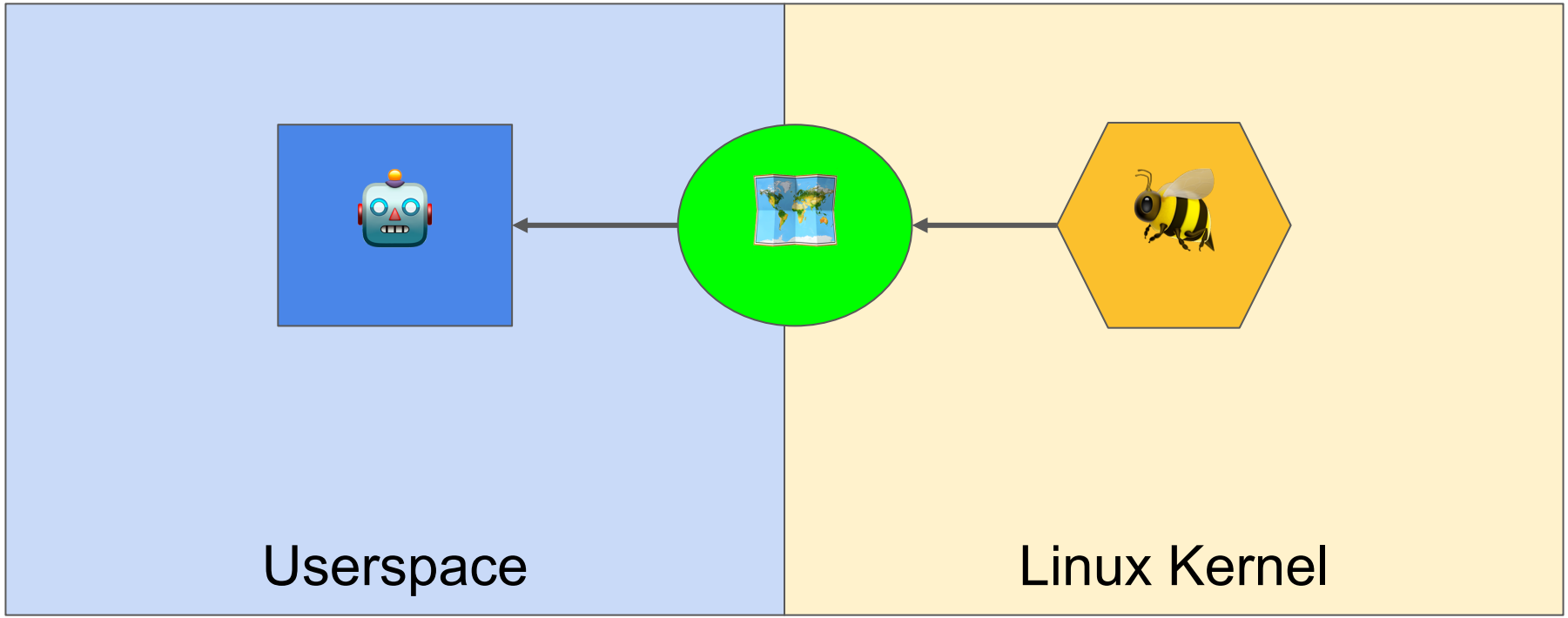
App v3: FizzBuzzFibonacci Feature

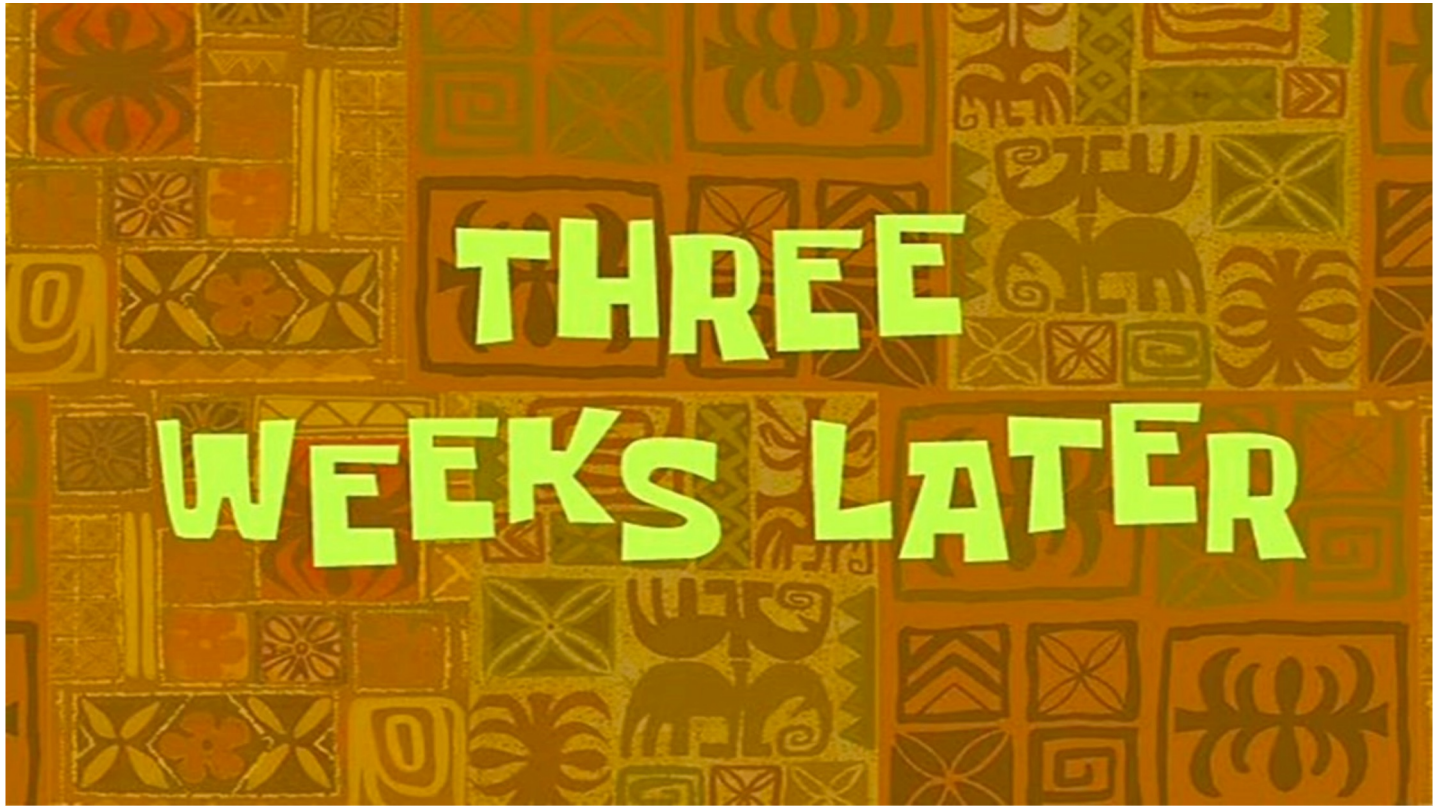


App v3: FizzBuzzFibonacci Feature



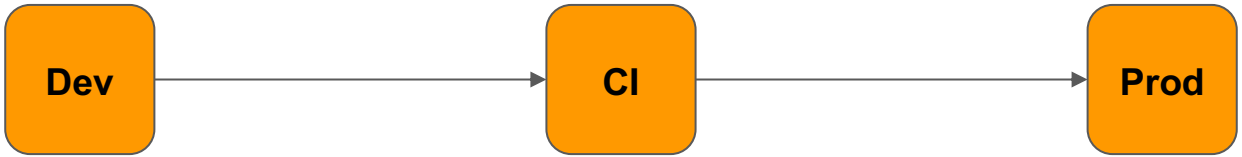
App v3: FizzBuzzFibonacci Feature



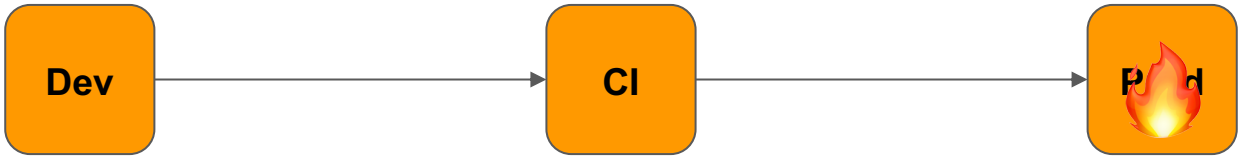


Bencher

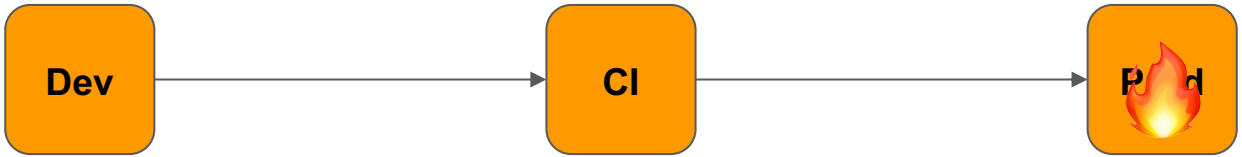
When do performance regression get detected?



When do performance regression get detected?



When do performance regression get detected?



App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = match (source_addr % 3, source_addr % 5) { ... };
    let opt_source_addr = is_fibonacci(source_addr as u8)
        .then_some(SourceAddr::Fibonacci)
        .or(match (source_addr % 3, source_addr % 5) {
            (0, 0) => Some(SourceAddr::FizzBuzz),
            (0, _) => Some(SourceAddr::Fizz),
            (_, 0) => Some(SourceAddr::Buzz),
            _ => None,
        });

    ...
    Ok(xdp_action::XDP_PASS)
}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = match (source_addr % 3, source_addr % 5) { ... };
    let opt_source_addr = is_fibonacci(source_addr as u8)
        .then_some(SourceAddr::Fibonacci)
        .or(match (source_addr % 3, source_addr % 5) {
            (0, 0) => Some(SourceAddr::FizzBuzz),
            (0, _) => Some(SourceAddr::Fizz),
            (_, 0) => Some(SourceAddr::Buzz),
            _ => None,
        });

    ...
    Ok(xdp_action::XDP_PASS)
}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn try_fun_xdp(ctx: &XdpContext) -> Result<u32, ()> {
    ...
    let opt_source_addr = match (source_addr % 3, source_addr % 5) { ... };
    let opt_source_addr = is_fibonacci(source_addr as u8)
        .then_some(SourceAddr::Fibonacci)
        .or(match (source_addr % 3, source_addr % 5) {
            (0, 0) => Some(SourceAddr::FizzBuzz),
            (0, _) => Some(SourceAddr::Fizz),
            (_, 0) => Some(SourceAddr::Buzz),
            _ => None,
        });

    ...
    Ok(xdp_action::XDP_PASS)
}
```

App v3: FizzBuzzFibonacci Feature

```rust
fn is_fibonacci(n: u8) -> bool {
    let (mut a, mut b) = (0, 1);
    while b < n {
        let c = a + b;
        a = b;
        b = c;
    }
    b == n
}
```

App v4: FizzBuzzFibonacci Memo Feature

```rust
fn is_fibonacci(n: u8) -> bool {
    let (mut a, mut b) = (0, 1);
    while b < n {
        let c = a + b;
        a = b;
        b = c;
    }
    b == n

}
```

App v4: FizzBuzzFibonacci Memo Feature

```rust
fn is_fibonacci(n: u8) -> bool {
    let (mut a, mut b) = (0, 1);
    while b < n {
        let c = a + b;
        a = b;
        b = c;
    }
    b == n

}
```

App v4: FizzBuzzFibonacci Memo Feature

```rust
fn is_fibonacci(n: u8) -> bool {
    let (mut a, mut b) = (0, 1);
    while b < n {
        let c = a + b;
        a = b;
        b = c;
    }
    b == n
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144)
}
```

App v4: FizzBuzzFibonacci Memo Feature

```rust
fn is_fibonacci(n: u8) -> bool {
    let (mut a, mut b) = (0, 1);
    while b < n {
        let c = a + b;
        a = b;
        b = c;
    }
    b == n
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144)
}
```

Give me a list of all fibonacci numbers below 256

Bencher

App v4: FizzBuzzFibonacci Memo Feature

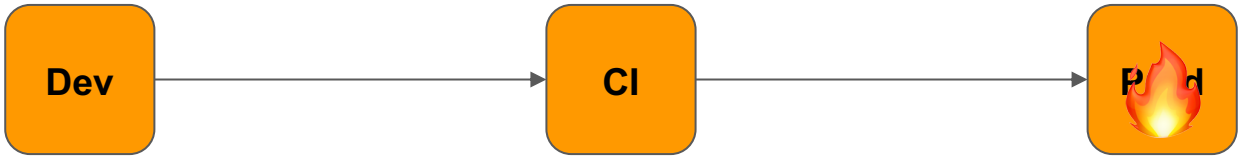
```rust
fn is_fibonacci(n: u8) -> bool {
    let (mut a, mut b) = (0, 1);
    while b < n {
        let c = a + b;
        a = b;
        b = c;
    }
    b == n
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

Give me a list of all fibonacci numbers below 256

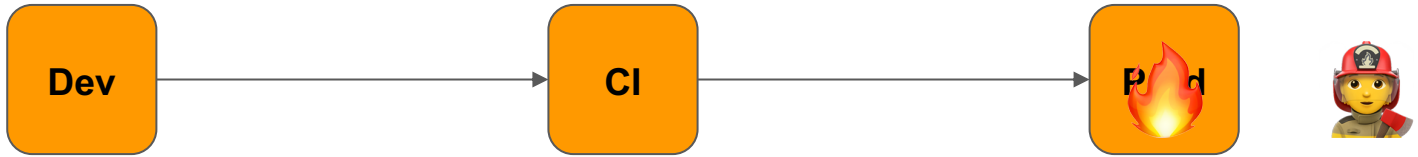
App v4: FizzBuzzFibonacci Memo Feature

```rust
fn is_fibonacci(n: u8) -> bool {
    let (mut a, mut b) = (0, 1);
    while b < n {
        let c = a + b;
        a = b;
        b = c;
    }
    b == n
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

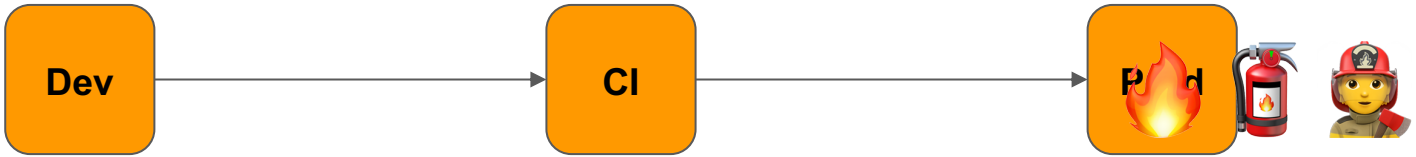
When do performance regression get detected?



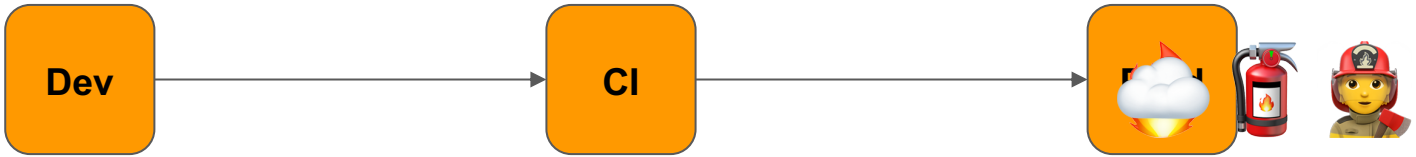
When do performance regression get detected?



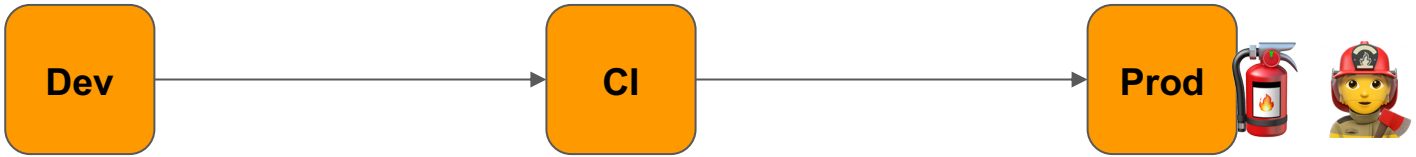
When do performance regression get detected?



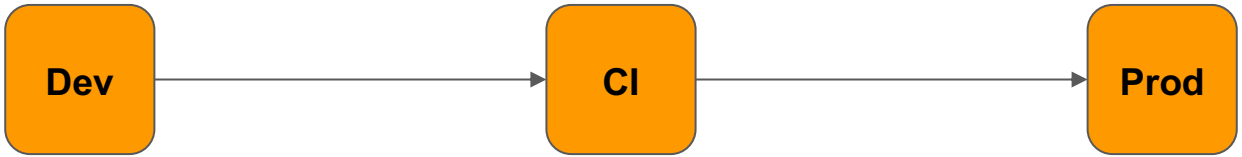
When do performance regression get detected?



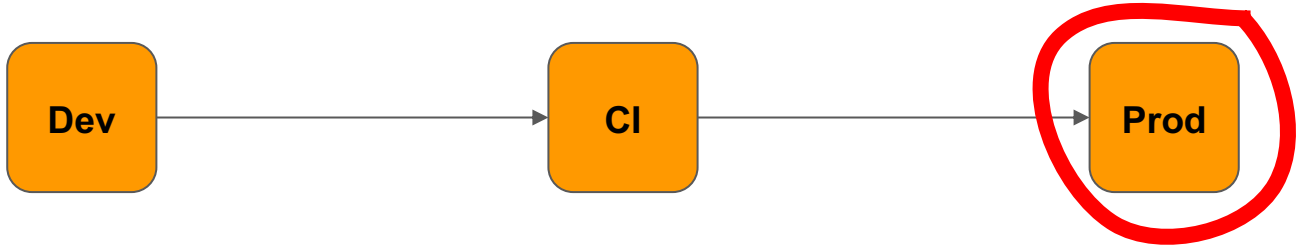
When do performance regression get detected?



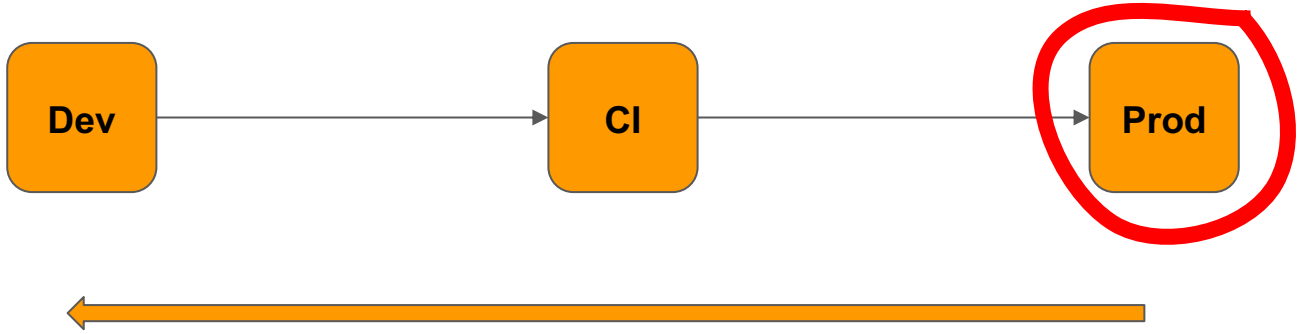
When do performance regression get detected?



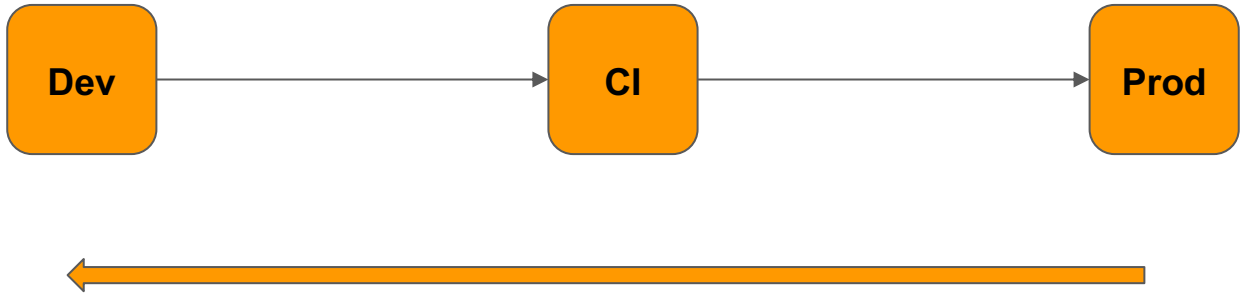
When do performance regression get detected?



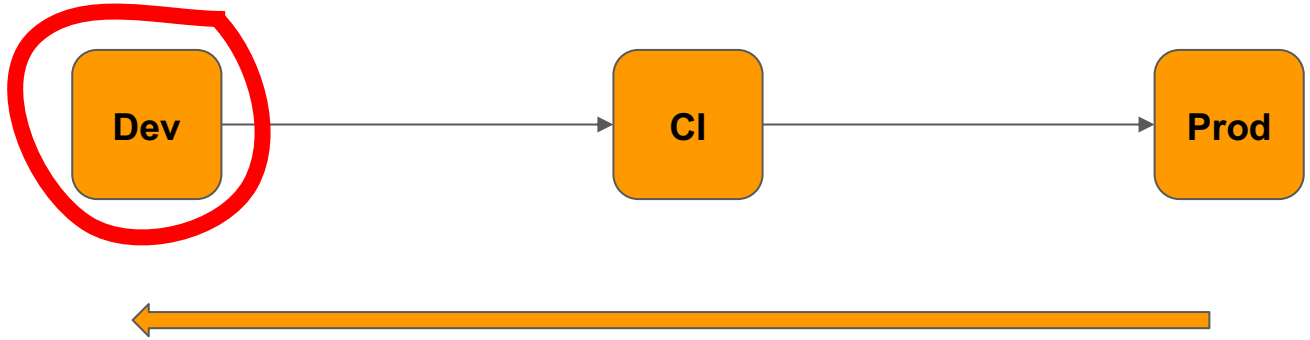
When do performance regression get detected?



When do performance regression get detected?



When do performance regression get detected?



Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust
- Benchmark eBPF program in Rust

You can't improve what you don't measure

Micro vs Macro Benchmarks

Micro vs Macro Benchmarks

Micro
(unit)

Micro vs Macro Benchmarks

Micro (unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

Micro vs Macro Benchmarks

Micro
(unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

Macro
(integration)

Micro vs Macro Benchmarks

Micro (unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

Macro (integration)

```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;

    loop { ... }
}
```

Micro vs Macro Benchmarks

Micro (unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
```

Macro (integration)

```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;

    loop { ... }
}
```

Bencher

Rust eBPF Micro-Benchmarking

Rust eBPF Micro-Benchmarking

- libtest bench

libtest bench

Bencher

libtest bench

- Nightly only

libtest bench

- Nightly only
- Use (unrelated) bencher crate for stable

libtest bench

- Nightly only
- Use (unrelated) bencher crate for stable
- Not actively developed

Rust eBPF Micro-Benchmarking

- libtest bench

Rust eBPF Micro-Benchmarking

- libtest bench
- Criterion

Criterion

Criterion

- Stable and nightly

Criterion

- Stable and nightly
- De facto standard in the community

Criterion

- Stable and nightly
- De facto standard in the community
- Much more feature rich

Rust eBPF Micro-Benchmarking

- libtest bench
- Criterion

Rust eBPF Micro-Benchmarking

- libtest bench
- Criterion
- Iai

lai

lai

- Stable and nightly but experimental

Iai

- Stable and nightly but experimental
- From the creator of Criterion

Iai

- Stable and nightly but experimental
- From the creator of Criterion
- Single-shot using Cachegrind
  - Instructions
  - L1 Accesses
  - L2 Accesses
  - RAM Accesses
  - Estimated Cycles

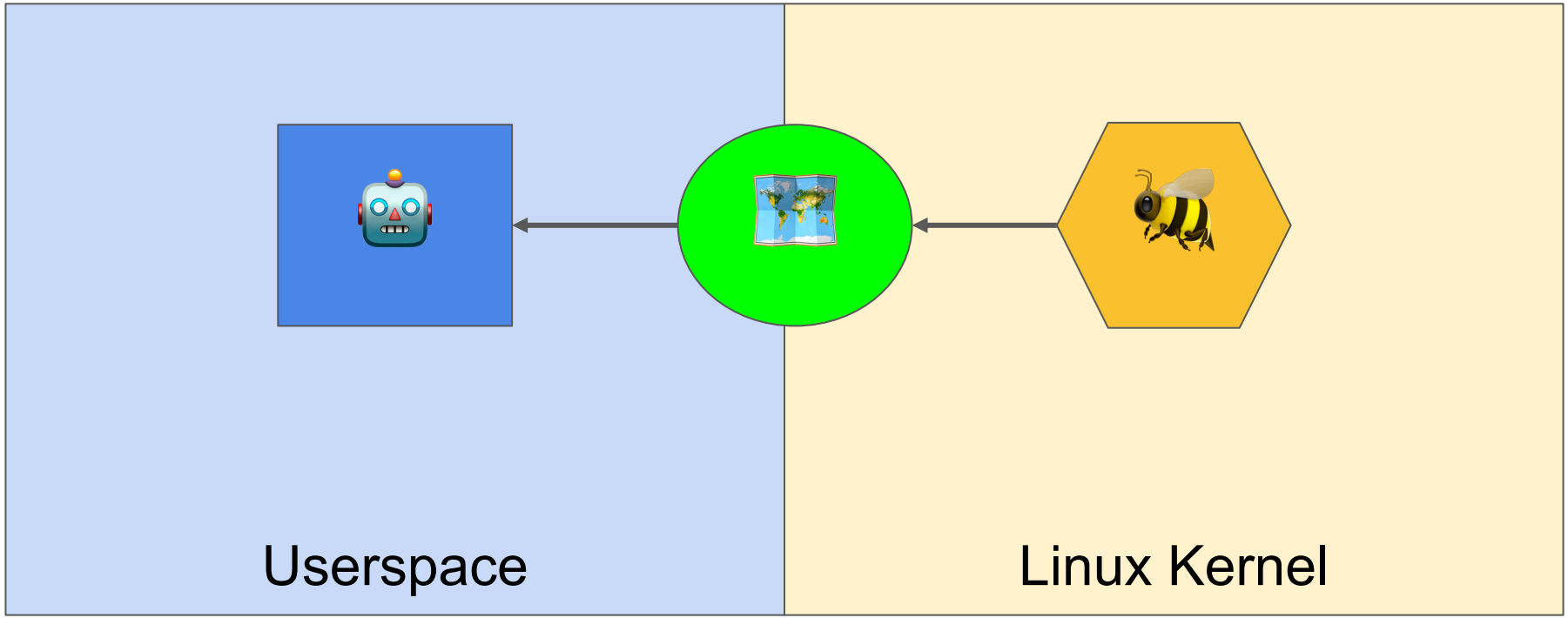
Rust eBPF Micro-Benchmarking

- libtest bench
- Criterion
- Iai

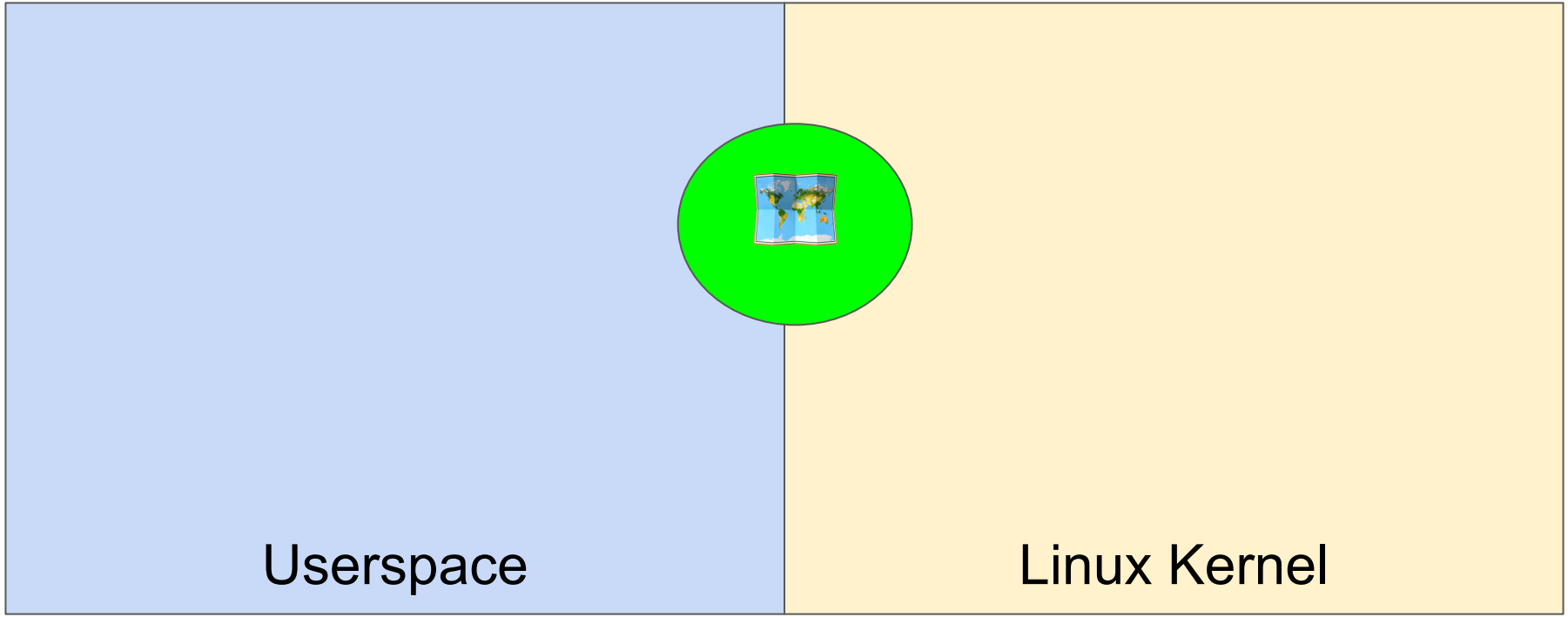
Rust eBPF Micro-Benchmarking

- libtest bench
- Criterion
- Iai

Rust eBPF Micro-Benchmarking

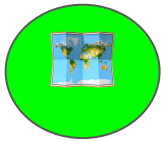


Rust eBPF Micro-Benchmarking



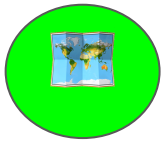
Micro-Benchmarking: Cargo.toml

Micro-Benchmarking: Cargo.toml



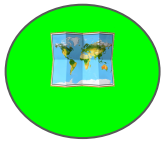
```toml
[dev-dependencies]
```

Micro-Benchmarking: Cargo.toml



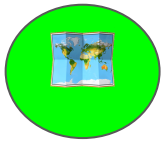
```toml
[dev-dependencies]
criterion = "0.4"
```

Micro-Benchmarking: Cargo.toml



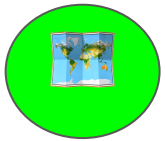
```toml
[dev-dependencies]
criterion = "0.4"

[[bench]]
```

Micro-Benchmarking: Cargo.toml



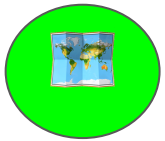
```toml
[dev-dependencies]
criterion = "0.4"

[[bench]]
name = "source_addr"
```

Micro-Benchmarking: Cargo.toml



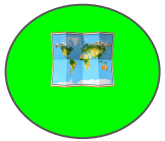
```toml
[dev-dependencies]
criterion = "0.4"

[[bench]]
name = "source_addr"
harness = false
```

Micro-Benchmarking: SourceAddr



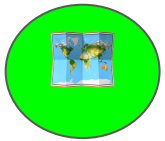
```
impl SourceAddr {









}
```

Micro-Benchmarking: SourceAddr



```rust
impl SourceAddr {
    pub fn new(source_addr: u32) -> Option<SourceAddr> {

    }
}
```

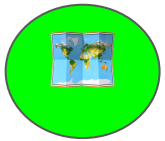
Micro-Benchmarking: SourceAddr



```rust
impl SourceAddr {
    pub fn new(source_addr: u32) -> Option<SourceAddr> {
        is_fibonacci(source_addr as u8)
            .then_some(SourceAddr::Fibonacci)
            .or(match (source_addr % 3, source_addr % 5) {
                (0, 0) => Some(SourceAddr::FizzBuzz),
                (0, _) => Some(SourceAddr::Fizz),
                (_, 0) => Some(SourceAddr::Buzz),
                _ => None,
            })
    }
}
```

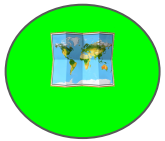
Micro-Benchmarking: benches/source_addr.rs

Micro-Benchmarking: benches/source_addr.rs



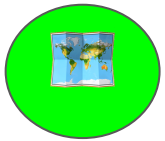
```rust
fn bench_source_addr(c: &mut Criterion) {









}
```

Micro-Benchmarking: benches/source_addr.rs



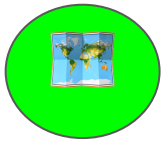
```rust
fn bench_source_addr(c: &mut Criterion) {
    c.bench_function("SourceAddr", |b| {
        b.iter(|| {


        })
    });
}
```

Micro-Benchmarking: benches/source_addr.rs



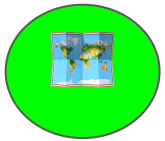
```rust
fn bench_source_addr(c: &mut Criterion) {
    c.bench_function("SourceAddr", |b| {
        b.iter(|| {
            for i in 0..256 {
                SourceAddr::new(i);
            }
        })
    });
}
```

Micro-Benchmarking: benches/source_addr.rs



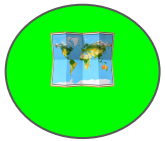
```rust
fn bench_source_addr(c: &mut Criterion) {
    c.bench_function("SourceAddr", |b| {
        b.iter(|| {
            for i in 0..256 {
                SourceAddr::new(i);
            }
        })
    });
}


criterion_group!(benches, bench_source_addr);
```

Micro-Benchmarking: benches/source_addr.rs



```rust
fn bench_source_addr(c: &mut Criterion) {
    c.bench_function("SourceAddr", |b| {
        b.iter(|| {
            for i in 0..256 {
                SourceAddr::new(i);
            }
        })
    });
}


criterion_group!(benches, bench_source_addr);
criterion_main!(benches);
```

Micro-Benchmarking: FizzBuzzFibonacci

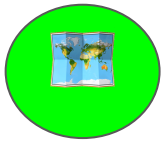


```
$ cargo bench
```

```
  4 (4.00%) high severe
```

Bencher

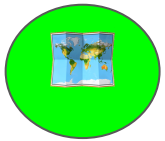
Micro-Benchmarking: FizzBuzzFibonacci



```
$ cargo bench
   Finished bench [optimized] target(s) in 1m 03s
     Running unittests src/lib.rs
(/home/epompeii/Code/bencher/examples/ebpf/target/release/deps/ebpf_common-23fb553d75cce271)
```

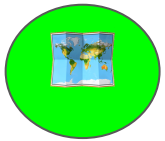
```
  4 (4.00%) high severe
```

Micro-Benchmarking: FizzBuzzFibonacci



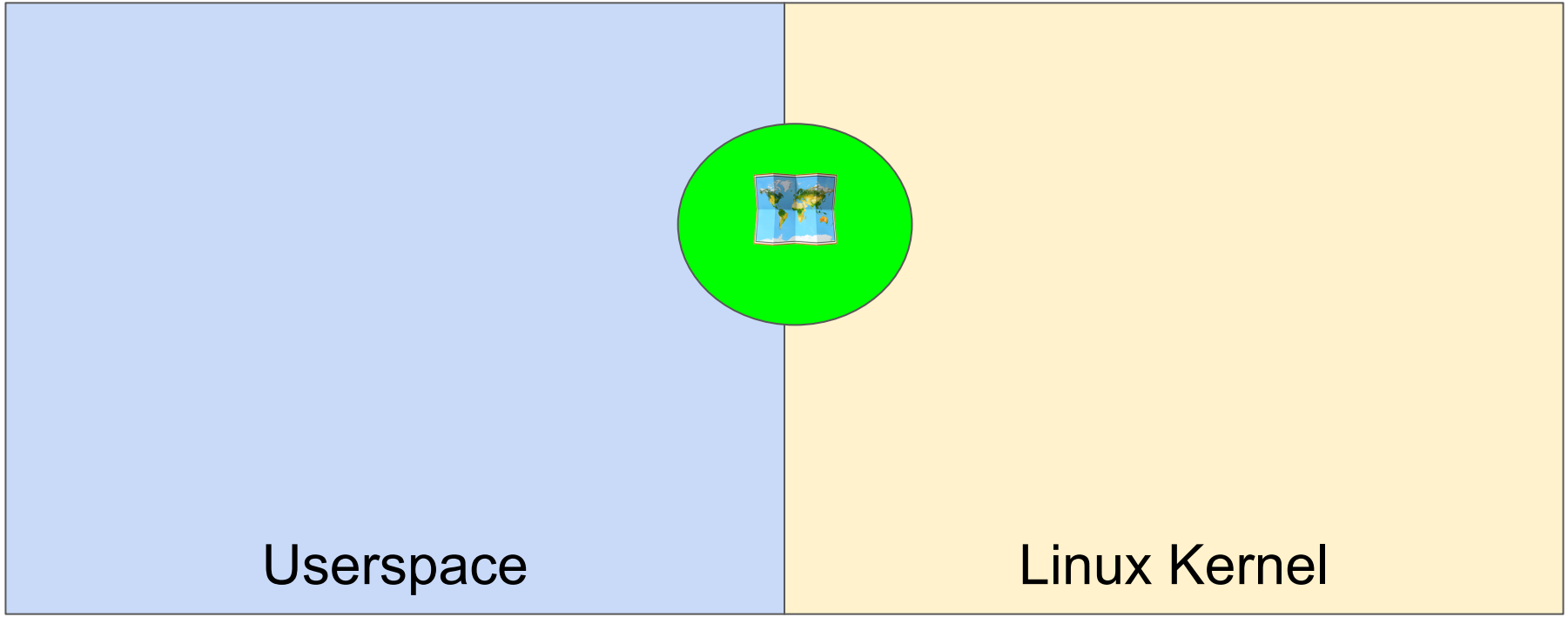
```
$ cargo bench
   Finished bench [optimized] target(s) in 1m 03s
     Running unittests src/lib.rs
(/home/epompeii/Code/bencher/examples/ebpf/target/release/deps/ebpf_common-23fb553d75cce271)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Micro-Benchmarking: FizzBuzzFibonacci

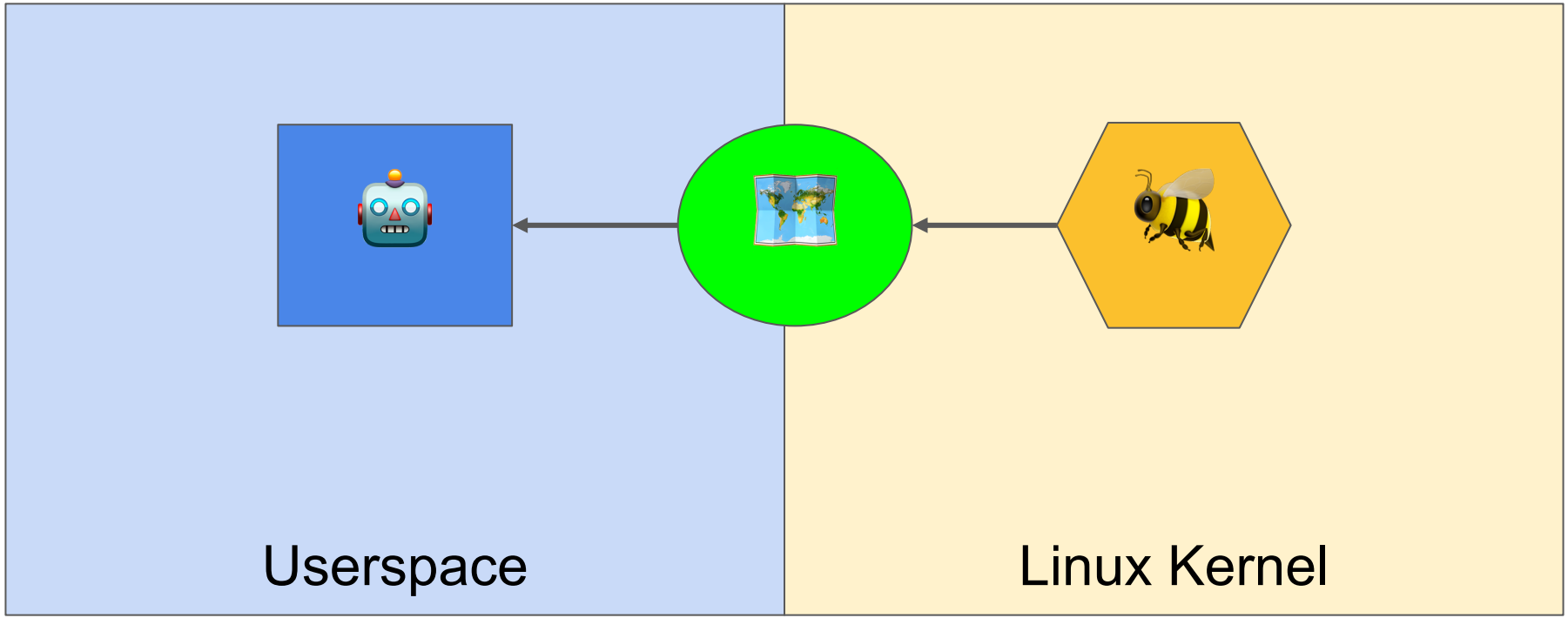


```
$ cargo bench
   Finished bench [optimized] target(s) in 1m 03s
     Running unittests src/lib.rs
(/home/epompeii/Code/bencher/examples/ebpf/target/release/deps/ebpf_common-23fb553d75cce271)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running benches/source_addr.rs
(/home/epompeii/Code/bencher/examples/ebpf/target/release/deps/source_addr-090d31387638198e)
SourceAddr              time:   [538.49 ns 540.24 ns 542.58 ns]
Found 13 outliers among 100 measurements (13.00%)
  9 (9.00%) high mild
  4 (4.00%) high severe
```

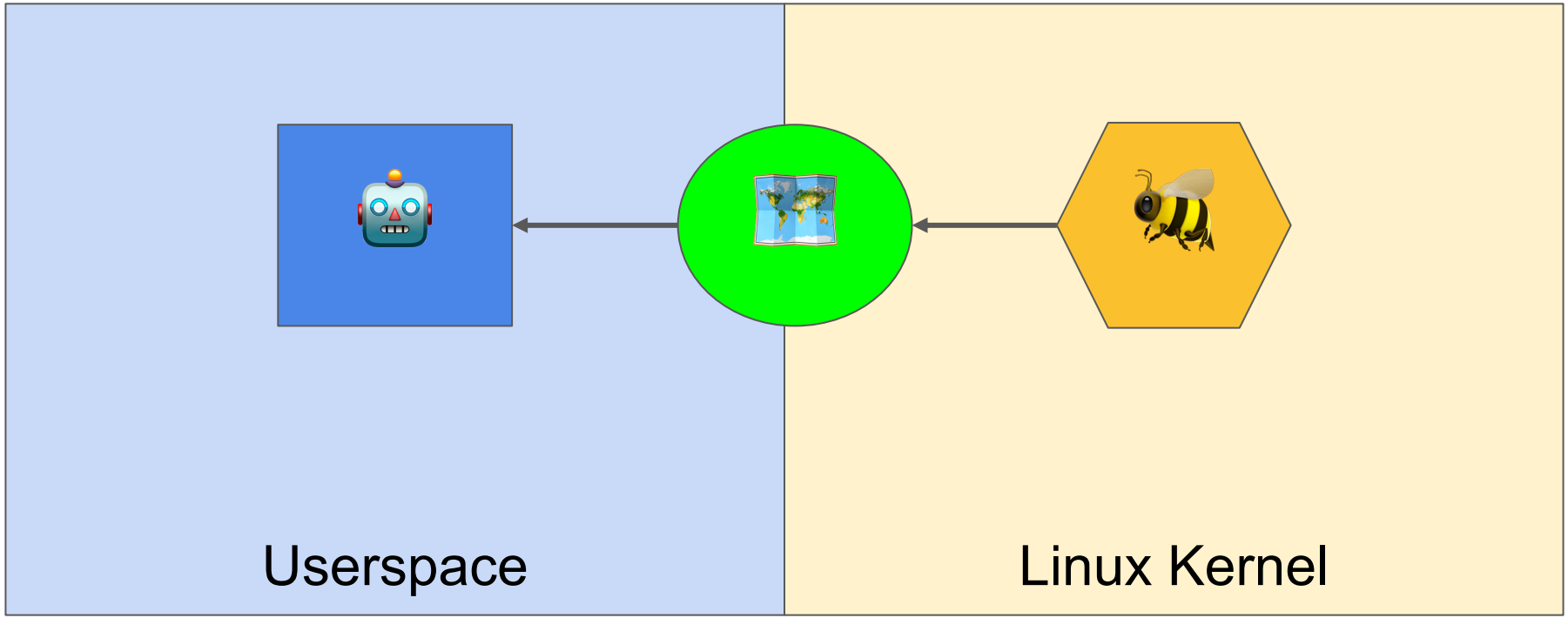
Rust eBPF Micro-Benchmarking



Rust eBPF Micro-Benchmarking



Rust eBPF Micro-Benchmarking



Micro vs Macro Benchmarks

Micro (unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
```

Macro (integration)

```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;

    loop { ... }
}
```

Bencher

Micro vs Macro Benchmarks

Micro (unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

Macro (integration)

```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;

    loop { ... }
}
```

Micro vs Macro Benchmarks

Micro (unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

Macro (integration)

```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;

    loop { ... }
}
```

Bencher

eBPF Macro-Benchmarking

eBPF Macro-Benchmarking

- kernel.bpf_stats_enabled

kernel.bpf_stats_enabled

kernel.bpf_stats_enabled

- Collects run_time_ns and run_cnt on all eBPF programs

kernel.bpf_stats_enabled

- Collects run_time_ns and run_cnt on all eBPF programs
- Added in v5.1

kernel.bpf_stats_enabled

- Collects run_time_ns and run_cnt on all eBPF programs
- Added in v5.1
- Off by default

eBPF Macro-Benchmarking

- kernel.bpf_stats_enabled

eBPF Macro-Benchmarking

- kernel.bpf_stats_enabled
- bpftool prog profile

bpftool prog profile

bpftool prog profile

- Collects hardware perf counts
  - Instructions
  - LLD Load
  - LLC Misses
  - Cycles

bpftool prog profile

- Collects hardware perf counts
  - Instructions
  - LLD Load
  - LLC Misses
  - Cycles
- Added in v5.7 but also requires bpftool built with clang >= 10.0.0

eBPF Macro-Benchmarking

- kernel.bpf_stats_enabled
- bpftool prog profile

eBPF Macro-Benchmarking

- kernel.bpf_stats_enabled
- bpftool prog profile
- bpftool prog run

bpftool prog run

bpftool prog run

- Run a specific eBPF program

bpftool prog run

- Run a specific eBPF program
- Provide input data and context* (except for XDP)

bpftool prog run

- Run a specific eBPF program
- Provide input data and context* (except for XDP)
- Returns output data and context

bpftool prog run

- Run a specific eBPF program
- Provide input data and context* (except for XDP)
- Returns output data and context
- Added in v4.12 but only for specific eBPF program types

bpftool prog run

- Run a specific eBPF program
- Provide input data and context* (except for XDP)
- Returns output data and context
- Added in v4.12 but only for specific eBPF program types
  - BPF_PROG_TYPE_CGROUP_SKB
  - BPF_PROG_TYPE_FLOW_DISSECTOR
  - BPF_PROG_TYPE_LWT_IN
  - BPF_PROG_TYPE_LWT_OUT
  - BPF_PROG_TYPE_LWT_SEG6LOCAL
  - BPF_PROG_TYPE_LWT_XMIT
  - BPF_PROG_TYPE_SCHED_ACT
  - BPF_PROG_TYPE_SCHED_CLS
  - BPF_PROG_TYPE_SOCKET_FILTER
  - BPF_PROG_TYPE_XDP

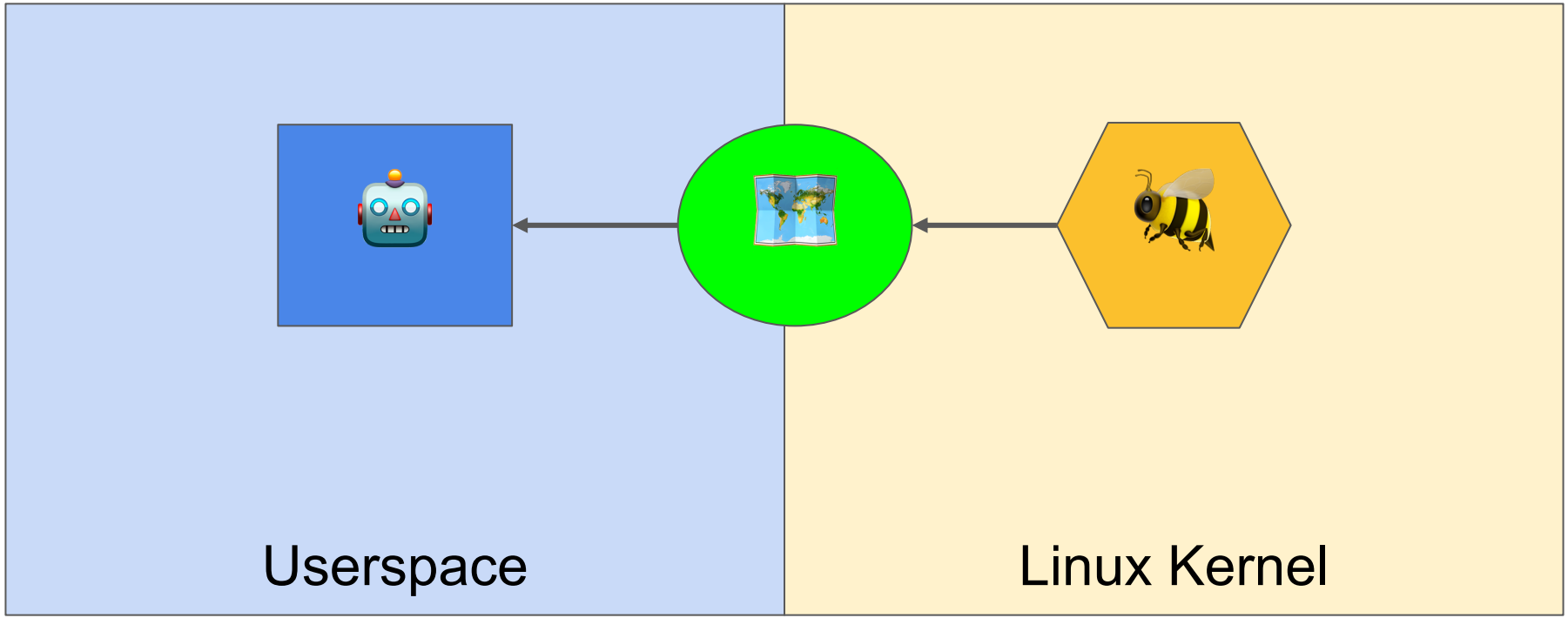
eBPF Macro-Benchmarking

- kernel.bpf_stats_enabled
- bpftool prog profile
- bpftool prog run

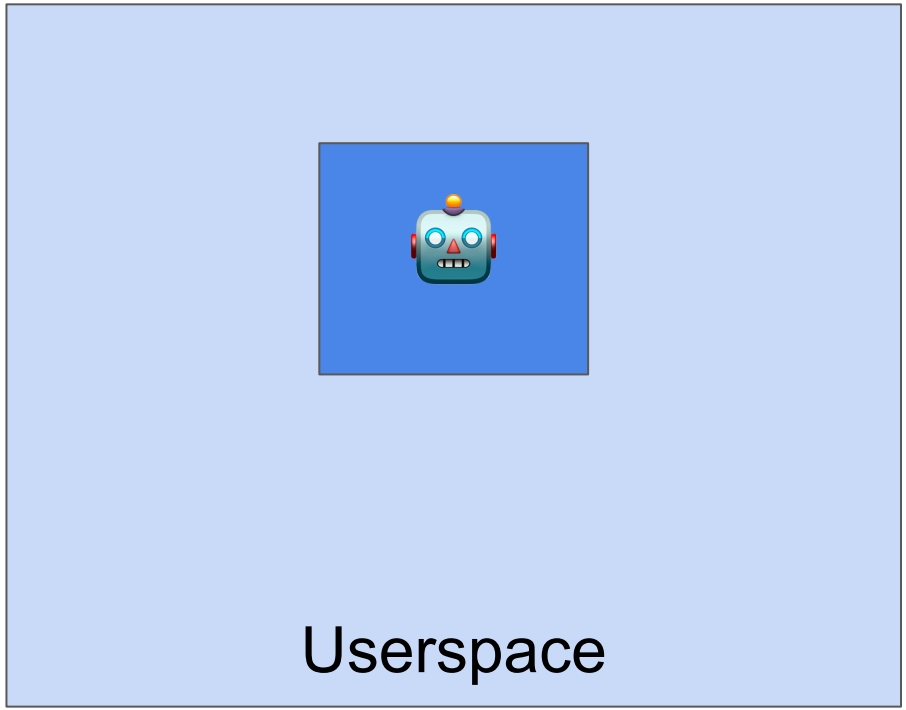
eBPF Macro-Benchmarking

- kernel.bpf_stats_enabled
- bpftool prog profile
- bpftool prog run

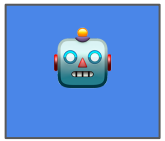
eBPF Macro-Benchmarking



eBPF Macro-Benchmarking

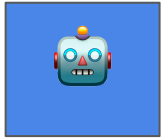


eBPF Macro-Benchmarking: New main



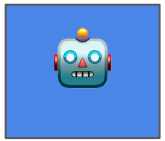
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
}
```

eBPF Macro-Benchmarking: New main



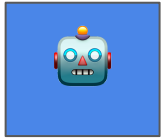
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();

    Ok(())
}
```

eBPF Macro-Benchmarking: New main



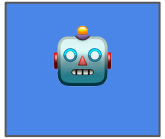
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();

    let shutdown = Arc::new(AtomicBool::new(false));

    Ok(())
}
```

eBPF Macro-Benchmarking: New main



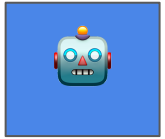
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();

    Ok(())
}
```

eBPF Macro-Benchmarking: New main



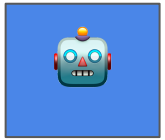
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    ebpf::run(&opt.iface, ebpf_shutdown).await?;

    Ok(())
}
```

eBPF Macro-Benchmarking: New main



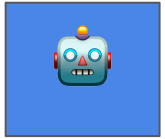
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    ebpf::run(&opt.iface, ebpf_shutdown).await?;

    info!("Waiting for Ctrl-C...");
    signal::ctrl_c().await?;

    Ok(())
}
```

eBPF Macro-Benchmarking: New main



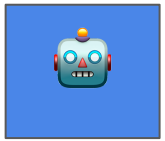
```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    ebpf::run(&opt.iface, ebpf_shutdown).await?;

    info!("Waiting for Ctrl-C...");
    signal::ctrl_c().await?;
    info!("Exiting...");
    shutdown.store(true, Ordering::Relaxed);

    Ok(())
}
```

eBPF Macro-Benchmarking: New main



```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let opt = Opt::parse();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    ebpf::run(&opt.iface, ebpf_shutdown).await?;

    info!("Waiting for Ctrl-C...");
    signal::ctrl_c().await?;
    info!("Exiting...");
    shutdown.store(true, Ordering::Relaxed);

    Ok(())
}
```

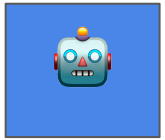
eBPF Macro-Benchmarking: New run



```
,
pub async fn run(iface: &str, shutdown: Arc<AtomicBool>) -> Result<Process, anyhow::Error> {
```

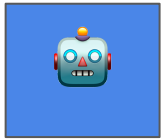
```
    ,,
}
```

eBPF Macro-Benchmarking: New run



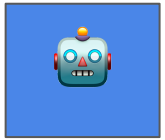
```rust
pub struct Process {
    pub pid: u32,
    pub prog_fd: i32,
    pub handle: tokio::task::JoinHandle<Result<(), anyhow::Error>>,
}
pub async fn run(iface: &str, shutdown: Arc<AtomicBool>) -> Result<Process, anyhow::Error> {









    ,,
}
```

eBPF Macro-Benchmarking: New run



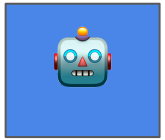
```rust
pub struct Process {
    pub pid: u32,
    pub prog_fd: i32,
    pub handle: tokio::task::JoinHandle<Result<(), anyhow::Error>>,
}
pub async fn run(iface: &str, shutdown: Arc<AtomicBool>) -> Result<Process, anyhow::Error> {
    let pid = std::process::id();









    ,,
}
```

eBPF Macro-Benchmarking: New run



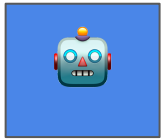
```rust
pub struct Process {
    pub pid: u32,
    pub prog_fd: i32,
    pub handle: tokio::task::JoinHandle<Result<(), anyhow::Error>>,
}
pub async fn run(iface: &str, shutdown: Arc<AtomicBool>) -> Result<Process, anyhow::Error> {
    let pid = std::process::id();
    let prog_fd = bpf.program("fun_xdp").unwrap().fd().unwrap().as_raw_fd();









    ,,
}
```

eBPF Macro-Benchmarking: New run



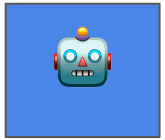
```
pub struct Process {
    pub pid: u32,
    pub prog_fd: i32,
    pub handle: tokio::task::JoinHandle<Result<(), anyhow::Error>>,
}
pub async fn run(iface: &str, shutdown: Arc<AtomicBool>) -> Result<Process, anyhow::Error> {
    let pid = std::process::id();
    let prog_fd = bpf.program("fun_xdp").unwrap().fd().unwrap().as_raw_fd();
    let handle = tokio::spawn(async move { spawn_agent(&mut bpf, shutdown).await });









    ,,
}
```

eBPF Macro-Benchmarking: New run



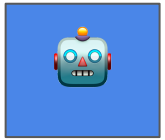
```rust
pub struct Process {
    pub pid: u32,
    pub prog_fd: i32,
    pub handle: tokio::task::JoinHandle<Result<(), anyhow::Error>>,
}
pub async fn run(iface: &str, shutdown: Arc<AtomicBool>) -> Result<Process, anyhow::Error> {
    let pid = std::process::id();
    let prog_fd = bpf.program("fun_xdp").unwrap().fd().unwrap().as_raw_fd();
    let handle = tokio::spawn(async move { spawn_agent(&mut bpf, shutdown).await });
    Ok(Process {
        pid,
        prog_fd,
        handle,
    })
}
```

eBPF Macro-Benchmarking: New spawn_agent



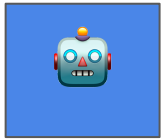
```rust
async fn spawn_agent(bpf: &mut Bpf, shutdown: Arc<AtomicBool>) -> Result<(), anyhow::Error> {









    Ok(())
}
```

eBPF Macro-Benchmarking: New spawn_agent



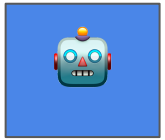
```rust
async fn spawn_agent(bpf: &mut Bpf, shutdown: Arc<AtomicBool>) -> Result<(), anyhow::Error> {
    let mut xdp_map: aya::maps::Queue<_, SourceAddr> =
        aya::maps::Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap()).unwrap();









    Ok(())
}
```

eBPF Macro-Benchmarking: New spawn_agent



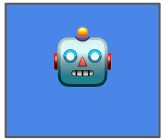
```rust
async fn spawn_agent(bpf: &mut Bpf, shutdown: Arc<AtomicBool>) -> Result<(), anyhow::Error> {
    let mut xdp_map: aya::maps::Queue<_, SourceAddr> =
        aya::maps::Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap()).unwrap();

    loop {
        while let Ok(source_addr) = xdp_map.pop(0) {
            info!("{:?}", source_addr);
        }

        ;
    }

    Ok(())
}
```

eBPF Macro-Benchmarking: New spawn_agent



```rust
async fn spawn_agent(bpf: &mut Bpf, shutdown: Arc<AtomicBool>) -> Result<(), anyhow::Error> {
    let mut xdp_map: aya::maps::Queue<_, SourceAddr> =
        aya::maps::Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap()).unwrap();

    loop {
        while let Ok(source_addr) = xdp_map.pop(0) {
            info!("{:?}", source_addr);
        }
        if shutdown.load(Ordering::Relaxed) {
            break;
        }
    }

    Ok(())
}
```

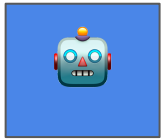
eBPF Macro-Benchmarking: New spawn_agent



```rust
async fn spawn_agent(bpf: &mut Bpf, shutdown: Arc<AtomicBool>) -> Result<(), anyhow::Error> {
    let mut xdp_map: aya::maps::Queue<_, SourceAddr> =
        aya::maps::Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap()).unwrap();

    loop {
        while let Ok(source_addr) = xdp_map.pop(0) {
            info!("{:?}", source_addr);
        }
        if shutdown.load(Ordering::Relaxed) {
            break;
        }
    }

    Ok(())
}
```

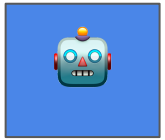
eBPF Macro-Benchmarking: Custom Harness

eBPF Macro-Benchmarking: Custom Harness



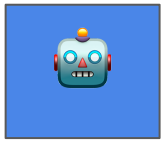
```rust
#[derive(Debug)]
pub struct EBpfBenchmark {
    pub name: &'static str,
    pub benchmark_fn: fn() -> f64,
}
```

eBPF Macro-Benchmarking: Custom Harness



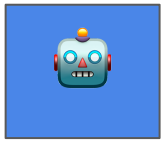
```rust
#[derive(Debug)]
pub struct EBpfBenchmark {
    pub name: &'static str,
    pub benchmark_fn: fn() -> f64,
}

inventory::collect!(EBpfBenchmark);
```

eBPF Macro-Benchmarking: Custom Harness



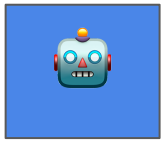
```
fn main() {

}
```

eBPF Macro-Benchmarking: Custom Harness



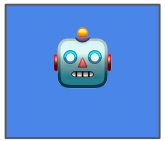
```rust
fn main() {
    let mut results = Vec::new();

}
```

eBPF Macro-Benchmarking: Custom Harness



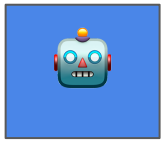
```rust
fn main() {
    let mut results = Vec::new();

    for benchmark in inventory::iter::<EBpfBenchmark> {

    }

}
```

eBPF Macro-Benchmarking: Custom Harness



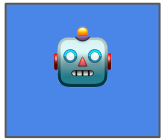
```rust
fn main() {
    let mut results = Vec::new();

    for benchmark in inventory::iter::<EBpfBenchmark> {
        let benchmark_name = benchmark.name.parse().unwrap();


    }




}
```

eBPF Macro-Benchmarking: Custom Harness



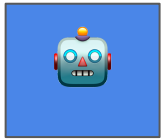
```
fn main() {
    let mut results = Vec::new();

    for benchmark in inventory::iter::<EBpfBenchmark> {
        let benchmark_name = benchmark.name.parse().unwrap();
        let ison metric = JsonMetric::new((benchmark.benchmark fn)(). None. None):
    }
}
```

eBPF Macro-Benchmarking: Custom Harness



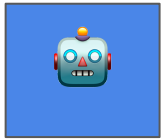
```rust
fn main() {
    let mut results = Vec::new();

    for benchmark in inventory::iter::<EBpfBenchmark> {
        let benchmark_name = benchmark.name.parse().unwrap();
        let json_metric = JsonMetric::new((benchmark.benchmark_fn)(), None, None);
        results.push((benchmark_name, json_metric));
    }

}
```

eBPF Macro-Benchmarking: Custom Harness



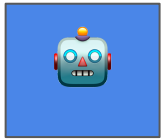
```rust
fn main() {
    let mut results = Vec::new();

    for benchmark in inventory::iter::<EBpfBenchmark> {
        let benchmark_name = benchmark.name.parse().unwrap();
        let json_metric = JsonMetric::new((benchmark.benchmark_fn)(), None, None);
        results.push((benchmark_name, json_metric));
    }

    let adapter_results = AdapterResults::new_latency(results).unwrap();



}
```

eBPF Macro-Benchmarking: Custom Harness



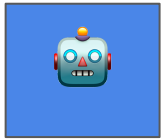
```rust
fn main() {
    let mut results = Vec::new();

    for benchmark in inventory::iter::<EBpfBenchmark> {
        let benchmark_name = benchmark.name.parse().unwrap();
        let json_metric = JsonMetric::new((benchmark.benchmark_fn)(), None, None);
        results.push((benchmark_name, json_metric));
    }

    let adapter_results = AdapterResults::new_latency(results).unwrap();
    let adapter_results_str = serde_json::to_string_pretty(&adapter_results).unwrap();


}
```

eBPF Macro-Benchmarking: Custom Harness



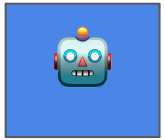
```rust
fn main() {
    let mut results = Vec::new();

    for benchmark in inventory::iter::<EBpfBenchmark> {
        let benchmark_name = benchmark.name.parse().unwrap();
        let json_metric = JsonMetric::new((benchmark.benchmark_fn)(), None, None);
        results.push((benchmark_name, json_metric));
    }

    let adapter_results = AdapterResults::new_latency(results).unwrap();
    let adapter_results_str = serde_json::to_string_pretty(&adapter_results).unwrap();
    let mut file = File::create("../target/results.json").unwrap();

}
```

eBPF Macro-Benchmarking: Custom Harness



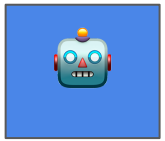
```rust
fn main() {
    let mut results = Vec::new();

    for benchmark in inventory::iter::<EBpfBenchmark> {
        let benchmark_name = benchmark.name.parse().unwrap();
        let json_metric = JsonMetric::new((benchmark.benchmark_fn)(), None, None);
        results.push((benchmark_name, json_metric));
    }

    let adapter_results = AdapterResults::new_latency(results).unwrap();
    let adapter_results_str = serde_json::to_string_pretty(&adapter_results).unwrap();
    let mut file = File::create("../target/results.json").unwrap();
    file.write_all(adapter_results_str.as_bytes()).unwrap();
}
```

eBPF Macro-Benchmarking: Custom Harness



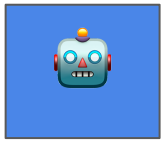
```rust
#[derive(Debug)]
pub struct EBpfBenchmark {
    pub name: &'static str,
    pub benchmark_fn: fn() -> f64,
}

inventory::collect!(EBpfBenchmark);
```

eBPF Macro-Benchmarking: Custom Harness



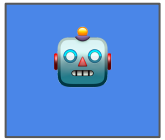
```rust
#[derive(Debug)]
pub struct EBpfBenchmark {
    pub name: &'static str,
    pub benchmark_fn: fn() -> f64,
}

inventory::collect!(EBpfBenchmark);
inventory::submit!(EBpfBenchmark {
    name: "fun_xdp",
    benchmark_fn: fun_xdp_benchmark
});
```

eBPF Macro-Benchmarking: Custom Harness



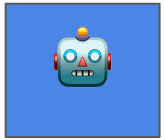
```
fn fun_xdp_benchmark() -> f64 {
}
```

eBPF Macro-Benchmarking: Custom Harness



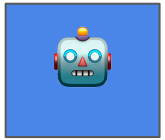
```
fn fun_xdp_benchmark() -> f64 {
    let rt = Runtime::new().unwrap();
}
```

eBPF Macro-Benchmarking: Custom Harness



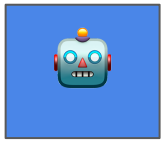
```rust
fn fun_xdp_benchmark() -> f64 {
    let rt = Runtime::new().unwrap();

    let shutdown = Arc::new(AtomicBool::new(false));

}
```

eBPF Macro-Benchmarking: Custom Harness



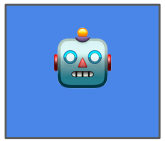
```rust
fn fun_xdp_benchmark() -> f64 {
    let rt = Runtime::new().unwrap();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();

    _
}
```

eBPF Macro-Benchmarking: Custom Harness



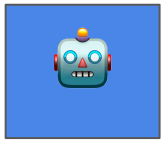
```rust
fn fun_xdp_benchmark() -> f64 {
    let rt = Runtime::new().unwrap();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    let process = rt.block_on(async { ebpf::run(IFACE, ebpf_shutdown).await.unwrap() });

    . _
}
```

eBPF Macro-Benchmarking: Custom Harness



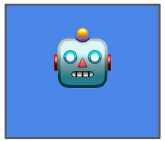
```rust
fn fun_xdp_benchmark() -> f64 {
    let rt = Runtime::new().unwrap();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    let process = rt.block_on(async { ebpf::run(IFACE, ebpf_shutdown).await.unwrap() });

    let _resp = rt.block_on(async { reqwest::get("https://bencher.dev").await.unwrap() });









    _
}
```

eBPF Macro-Benchmarking: Custom Harness



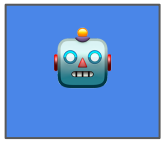
```rust
fn fun_xdp_benchmark() -> f64 {
    let rt = Runtime::new().unwrap();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    let process = rt.block_on(async { ebpf::run(IFACE, ebpf_shutdown).await.unwrap() });

    let _resp = rt.block_on(async { reqwest::get("https://bencher.dev").await.unwrap() });

    let bpf_stats = get_bpf_stats(&process);



    ' _
}
```

eBPF Macro-Benchmarking: Custom Harness



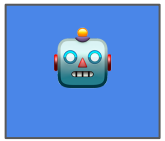
```rust
fn fun_xdp_benchmark() -> f64 {
    let rt = Runtime::new().unwrap();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    let process = rt.block_on(async { ebpf::run(IFACE, ebpf_shutdown).await.unwrap() });

    let _resp = rt.block_on(async { reqwest::get("https://bencher.dev").await.unwrap() });

    let bpf_stats = get_bpf_stats(&process);

    shutdown.store(true, Ordering::Relaxed);


}
```

eBPF Macro-Benchmarking: Custom Harness



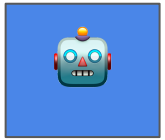
```rust
fn fun_xdp_benchmark() -> f64 {
    let rt = Runtime::new().unwrap();

    let shutdown = Arc::new(AtomicBool::new(false));
    let ebpf_shutdown = shutdown.clone();
    let process = rt.block_on(async { ebpf::run(IFACE, ebpf_shutdown).await.unwrap() });

    let _resp = rt.block_on(async { reqwest::get("https://bencher.dev").await.unwrap() });

    let bpf_stats = get_bpf_stats(&process);

    shutdown.store(true, Ordering::Relaxed);

    bpf_stats
}
```

eBPF Macro-Benchmarking: Custom Harness



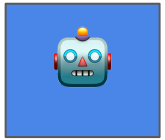
```
fn get_bpf_stats(process: &Process) -> f64 {









}
```

eBPF Macro-Benchmarking: Custom Harness



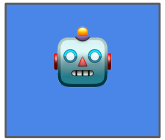
```rust
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();









}
```

eBPF Macro-Benchmarking: Custom Harness



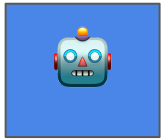
```rust
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();
    let reader = BufReader::new(fd_info);









}
```

eBPF Macro-Benchmarking: Custom Harness



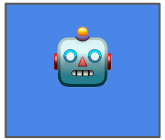
```rust
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();
    let reader = BufReader::new(fd_info);
    let (mut run_time_ns, mut run_ctn) = (None, None);









}
```

eBPF Macro-Benchmarking: Custom Harness



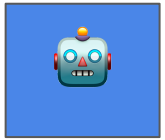
```
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();
    let reader = BufReader::new(fd_info);
    let (mut run_time_ns, mut run_ctn) = (None, None);
    for line in reader.lines().flatten() {









        ,
    }









}
```

eBPF Macro-Benchmarking: Custom Harness



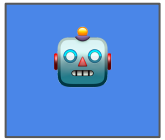
```
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();
    let reader = BufReader::new(fd_info);
    let (mut run_time_ns, mut run_ctn) = (None, None);
    for line in reader.lines().flatten() {
        if let Some(l) = line.strip_prefix("run_time_ns:") {
            run_time_ns = l.trim().parse::<u64>().ok();
        }



    }




}
```

eBPF Macro-Benchmarking: Custom Harness



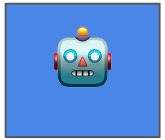
```rust
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();
    let reader = BufReader::new(fd_info);
    let (mut run_time_ns, mut run_ctn) = (None, None);
    for line in reader.lines().flatten() {
        if let Some(l) = line.strip_prefix("run_time_ns:") {
            run_time_ns = l.trim().parse::<u64>().ok();
        } else if let Some(l) = line.strip_prefix("run_cnt:") {
            run_ctn = l.trim().parse::<u64>().ok();
        }
    }



}
```

eBPF Macro-Benchmarking: Custom Harness



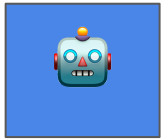
```rust
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();
    let reader = BufReader::new(fd_info);
    let (mut run_time_ns, mut run_ctn) = (None, None);
    for line in reader.lines().flatten() {
        if let Some(l) = line.strip_prefix("run_time_ns:") {
            run_time_ns = l.trim().parse::<u64>().ok();
        } else if let Some(l) = line.strip_prefix("run_cnt:") {
            run_ctn = l.trim().parse::<u64>().ok();
        }
    }
    match (run_time_ns, run_ctn) {



    }
}
```

eBPF Macro-Benchmarking: Custom Harness



```rust
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();
    let reader = BufReader::new(fd_info);
    let (mut run_time_ns, mut run_ctn) = (None, None);
    for line in reader.lines().flatten() {
        if let Some(l) = line.strip_prefix("run_time_ns:") {
            run_time_ns = l.trim().parse::<u64>().ok();
        } else if let Some(l) = line.strip_prefix("run_cnt:") {
            run_ctn = l.trim().parse::<u64>().ok();
        }
    }
    match (run_time_ns, run_ctn) {
        (Some(run_time_ns), Some(run_ctn)) if run_ctn != 0 => run_time_ns as f64 / run_ctn as f64,


    }
}
```

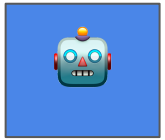
eBPF Macro-Benchmarking: Custom Harness



```rust
fn get_bpf_stats(process: &Process) -> f64 {
    let fd_info = File::open(format!("/proc/{}/fdinfo/{}", process.pid, process.prog_fd)).unwrap();
    let reader = BufReader::new(fd_info);
    let (mut run_time_ns, mut run_ctn) = (None, None);
    for line in reader.lines().flatten() {
        if let Some(l) = line.strip_prefix("run_time_ns:") {
            run_time_ns = l.trim().parse::<u64>().ok();
        } else if let Some(l) = line.strip_prefix("run_cnt:") {
            run_ctn = l.trim().parse::<u64>().ok();
        }
    }
    match (run_time_ns, run_ctn) {
        (Some(run_time_ns), Some(run_ctn)) if run_ctn != 0 => run_time_ns as f64 / run_ctn as f64,
        _ => 0.0,
    }
}
```

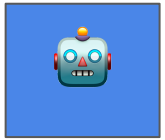
eBPF Macro-Benchmarking: Run it!

eBPF Macro-Benchmarking: Run it!



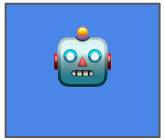
```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
```

eBPF Macro-Benchmarking: Run it!



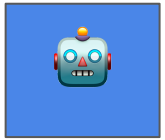
```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
```

eBPF Macro-Benchmarking: Run it!



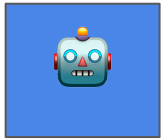
```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
$ cargo xtask build-ebpf --release
```

eBPF Macro-Benchmarking: Run it!



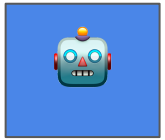
```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
$ cargo xtask build-ebpf --release
...
```

eBPF Macro-Benchmarking: Run it!



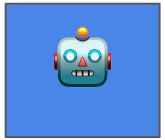
```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
$ cargo xtask build-ebpf --release
...
$ cd userspace
```

eBPF Macro-Benchmarking: Run it!



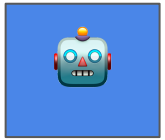
```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
$ cargo xtask build-ebpf --release
...
$ cd userspace
$ sudo -E $(which cargo) bench
```

eBPF Macro-Benchmarking: Run it!



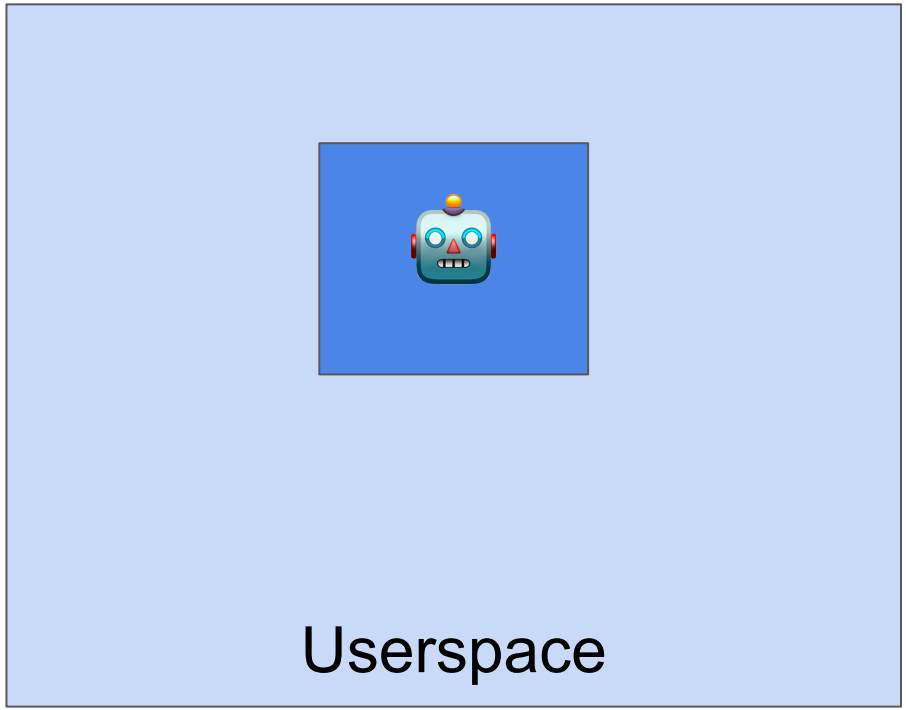
```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
$ cargo xtask build-ebpf --release
...
$ cd userspace
$ sudo -E $(which cargo) bench
...
```

eBPF Macro-Benchmarking: Run it!

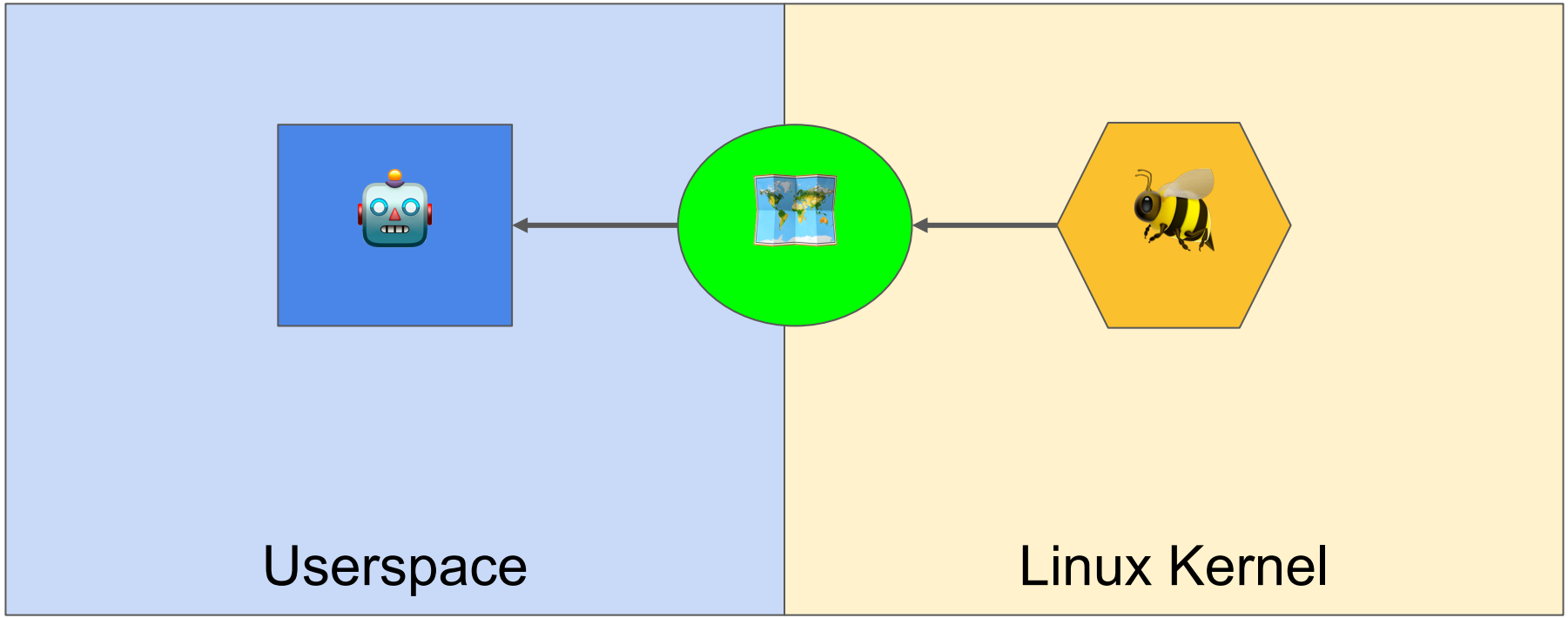


```
$ sudo -E $(which cargo) bench
   Finished bench [optimized] target(s) in 0.13s


   ...


     Running benches/xdp.rs (/home/epompeii/Code/bencher/examples/ebpf/target/release/deps/xdp-
d7c85fd4c85d089e)
{
  "fun_xdp": {
    "latency": {
      "value": 18249.727272727272,
      "lower_value": null,
      "upper_value": null
    }
  }
}
```

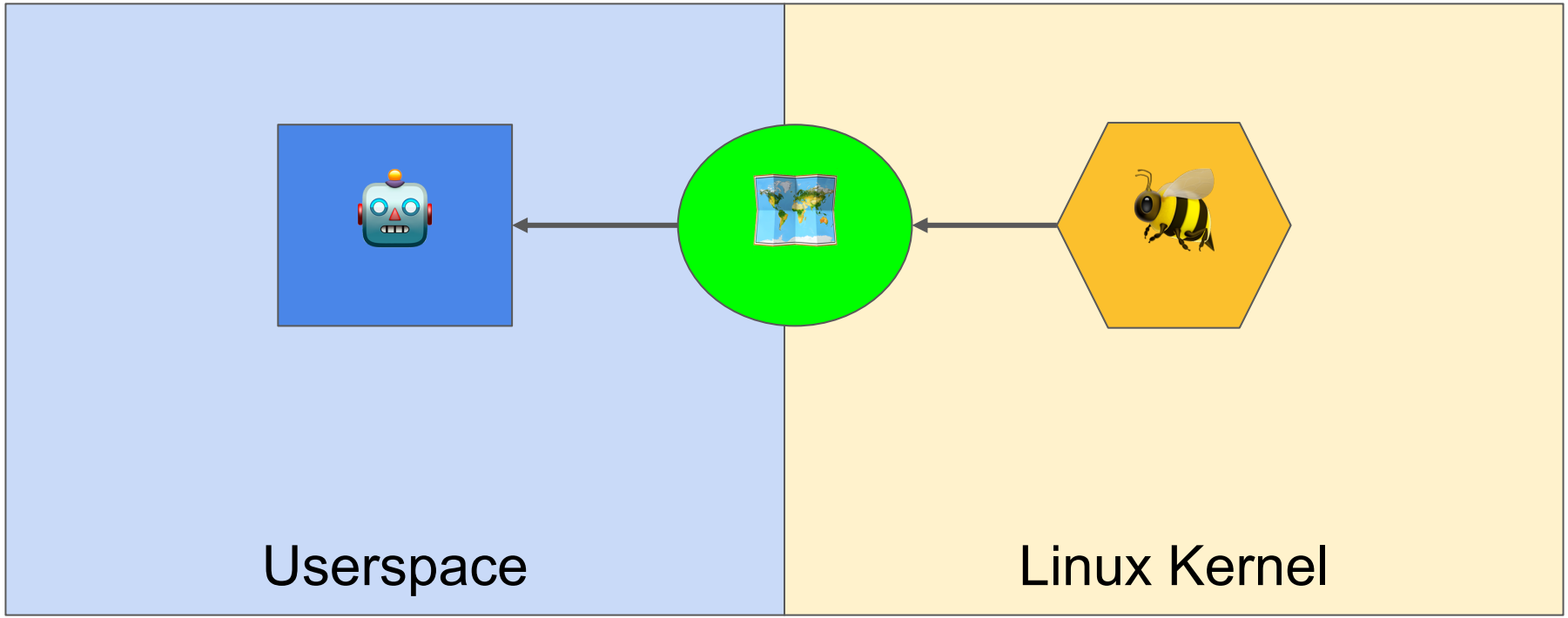
eBPF Macro-Benchmarking



eBPF Macro-Benchmarking



eBPF Macro-Benchmarking



Micro vs Macro Benchmarks

Micro (unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

Macro (integration)

```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;

    loop { ... }
}
```

Bencher

Micro vs Macro Benchmarks

Micro (unit)

```rust
fn is_fibonacci(n: u8) -> bool {
    matches!(n, 0 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233)
}
```

Macro (integration)

```rust
async fn spawn_agent(bpf: &mut Bpf) -> Result<(), anyhow::Error> {
    let mut xdp_map: Queue<_, SourceAddr> =
        Queue::try_from(bpf.map_mut("SOURCE_ADDR_QUEUE").unwrap())?;

    loop { ... }
}
```

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust
- Benchmark eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust
- Benchmark eBPF program in Rust
- Continuous Benchmarking eBPF program in Rust

Does software performance matter?

Software where performance matters

Software where performance matters

Software where performance matters

Software where performance matters

Performance Criticality ➡️

Software where performance matters

Software where performance matters

Software where performance matters

Software where performance matters

Bencher

Software where performance matters

Software where performance matters

✅ Software where performance matters

How to track software performance?

Not all software has access to production

Not all software has access to production

Not all software has access to production

Not all software has access to production

Access to Production ➡️

Not all software has access to production

Access to Production ➡️

Not all software has access to production

Not all software has access to production

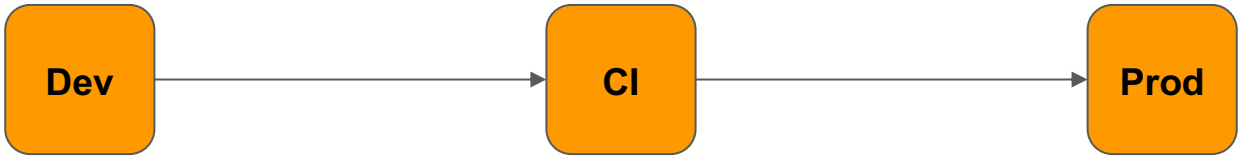
Not all software has access to production

Not all software has access to production

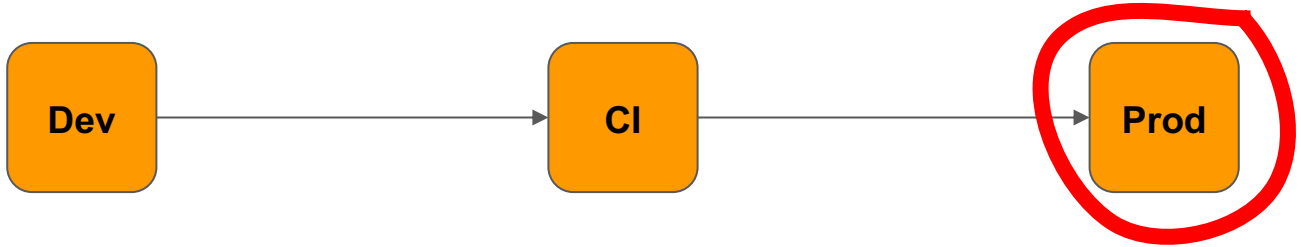
✅ Not all software has access to production

Bencher

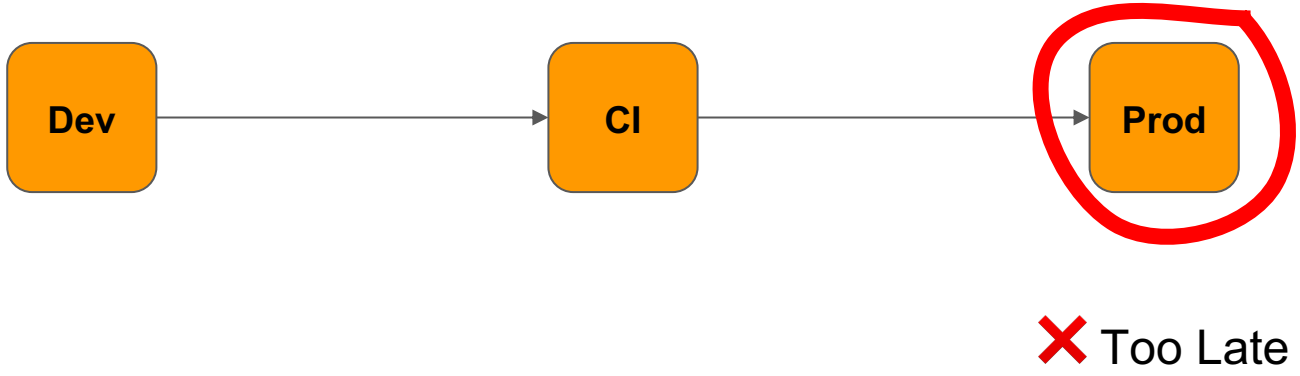
When do performance regression get detected?



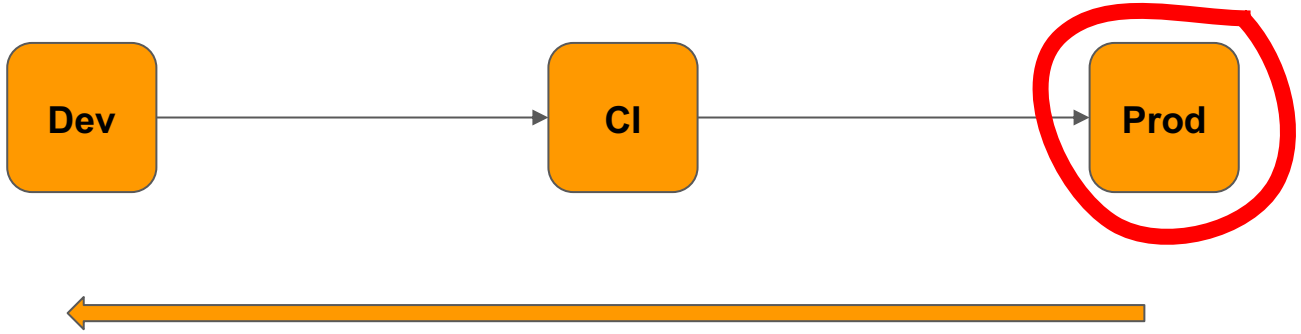
When do performance regression get detected?



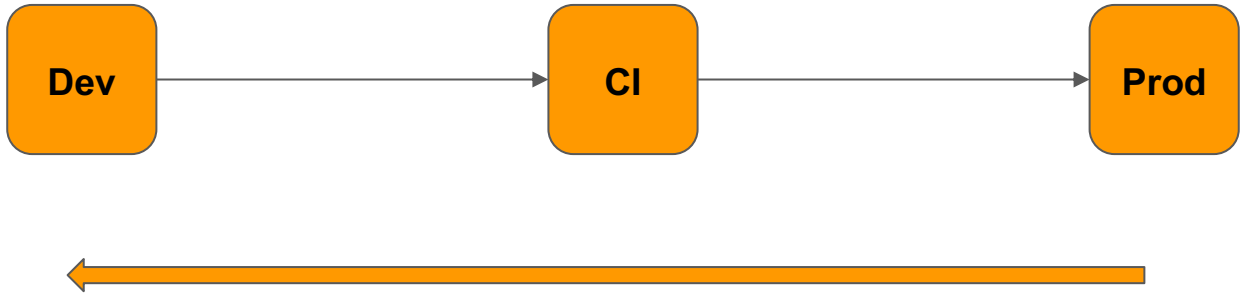
When do performance regression get detected?



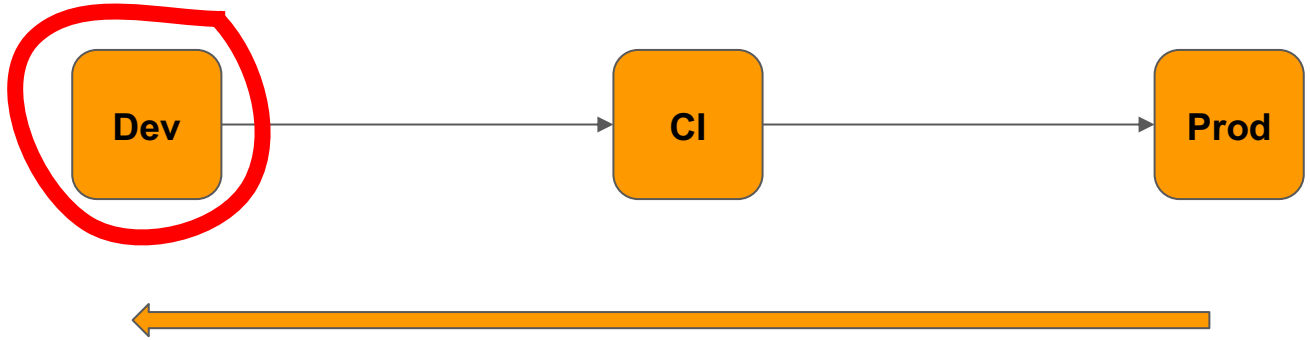
When do performance regression get detected?



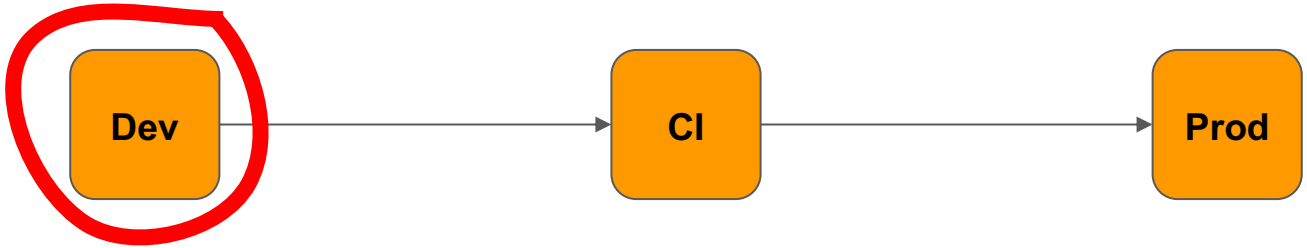
When do performance regression get detected?



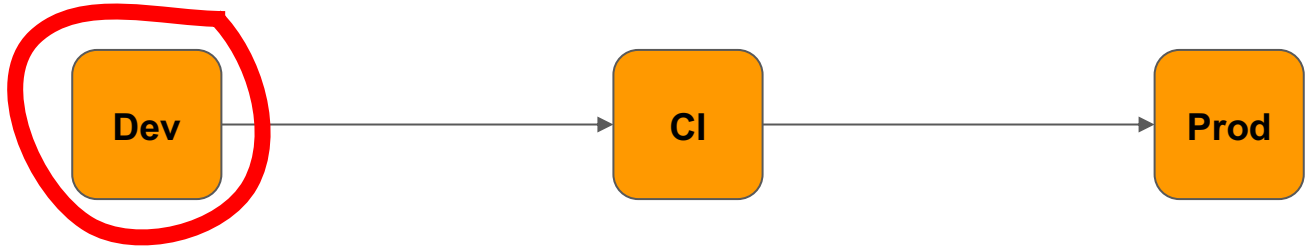
When do performance regression get detected?



When do performance regression get detected?

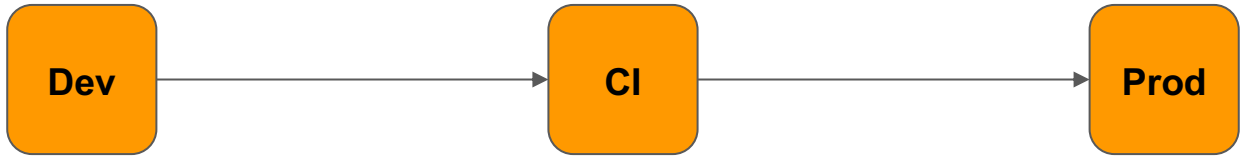


When do performance regression get detected?



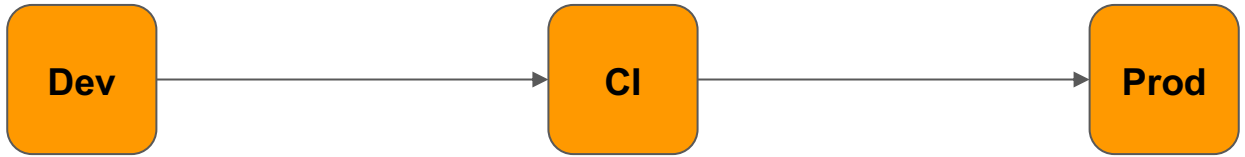
❌ Local Only

When do performance regression get detected?



❌ Local Only

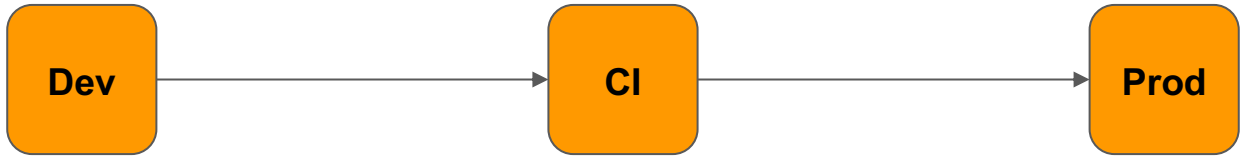
When do performance regression get detected?



❌ Local Only

❌ Too Late

When do performance regression get detected?



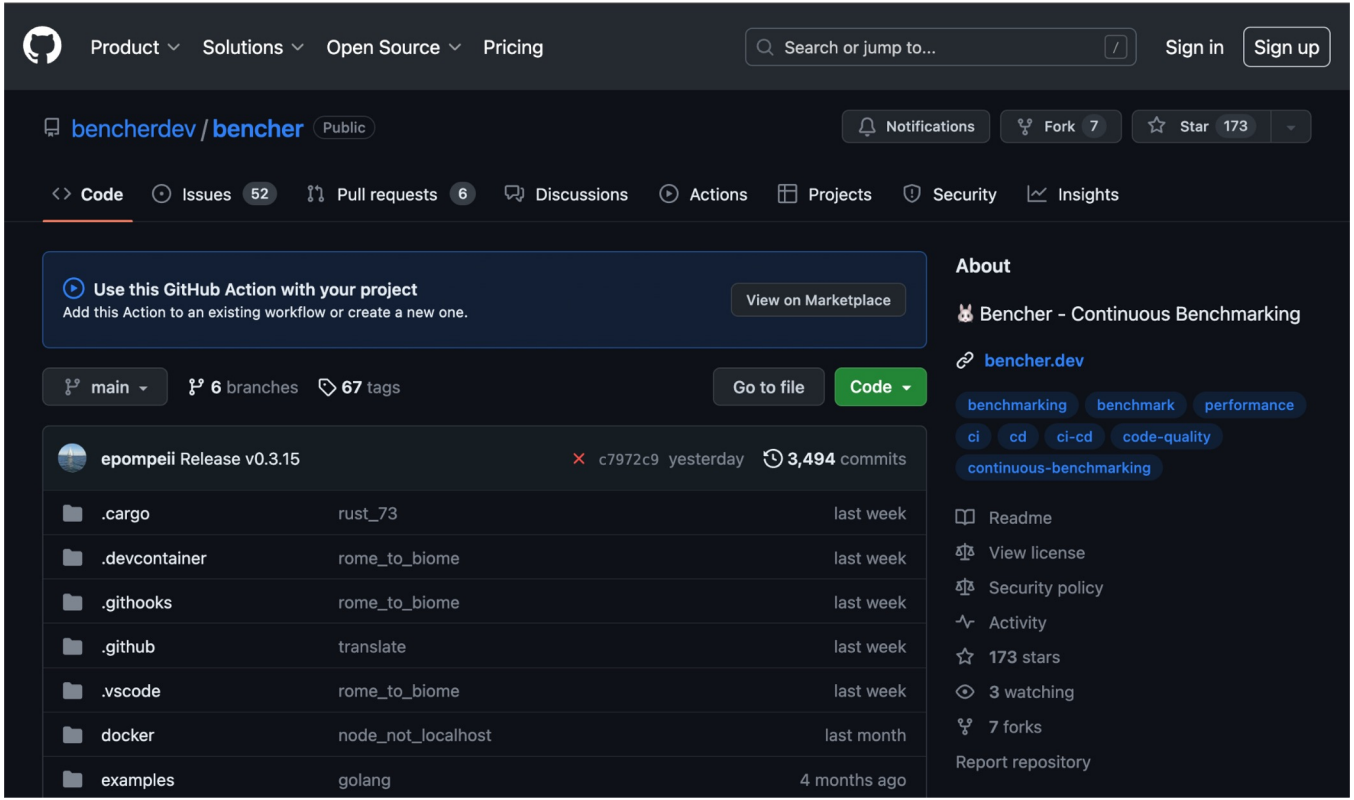
❌ Local Only

✅ Continuous Benchmarking

❌ Too Late

Bencher

Bencher


Product
Solutions
Open Source
Pricing
Search or jump to...
Sign in
Sign up
bencherdev / bencher
Public
Notifications
Fork 7
Star 173
Code
Issues 52
Pull requests 6
Discussions
Actions
Projects
Security
Insights
Use this GitHub Action with your project
Add this Action to an existing workflow or create a new one.
View on Marketplace
main
6 branches
67 tags
Go to file
Code
epompeii Release v0.3.15
c7972c9 yesterday
3,494 commits
.cargo rust_73 last week
.devcontainer rome_to_biome last week
.githooks rome_to_biome last week
.github translate last week
.vscode rome_to_biome last week
docker node_not_localhost last month
examples golang 4 months ago
About
Bencher - Continuous Benchmarking
bencher.dev
benchmarking
benchmark
performance
ci
cd
ci-cd
code-quality
continuous-benchmarking
Readme
View license
Security policy
Activity
173 stars
3 watching
7 forks
Report repository

Bencher

Bencher

- Open Source

Bencher

- Open Source
- Self-Hosted & SaaS

Bencher

- Open Source
- Self-Hosted & SaaS
- Multitenancy

Bencher

- Open Source
- Self-Hosted & SaaS
- Multitenancy
- Multi-Language

Bencher

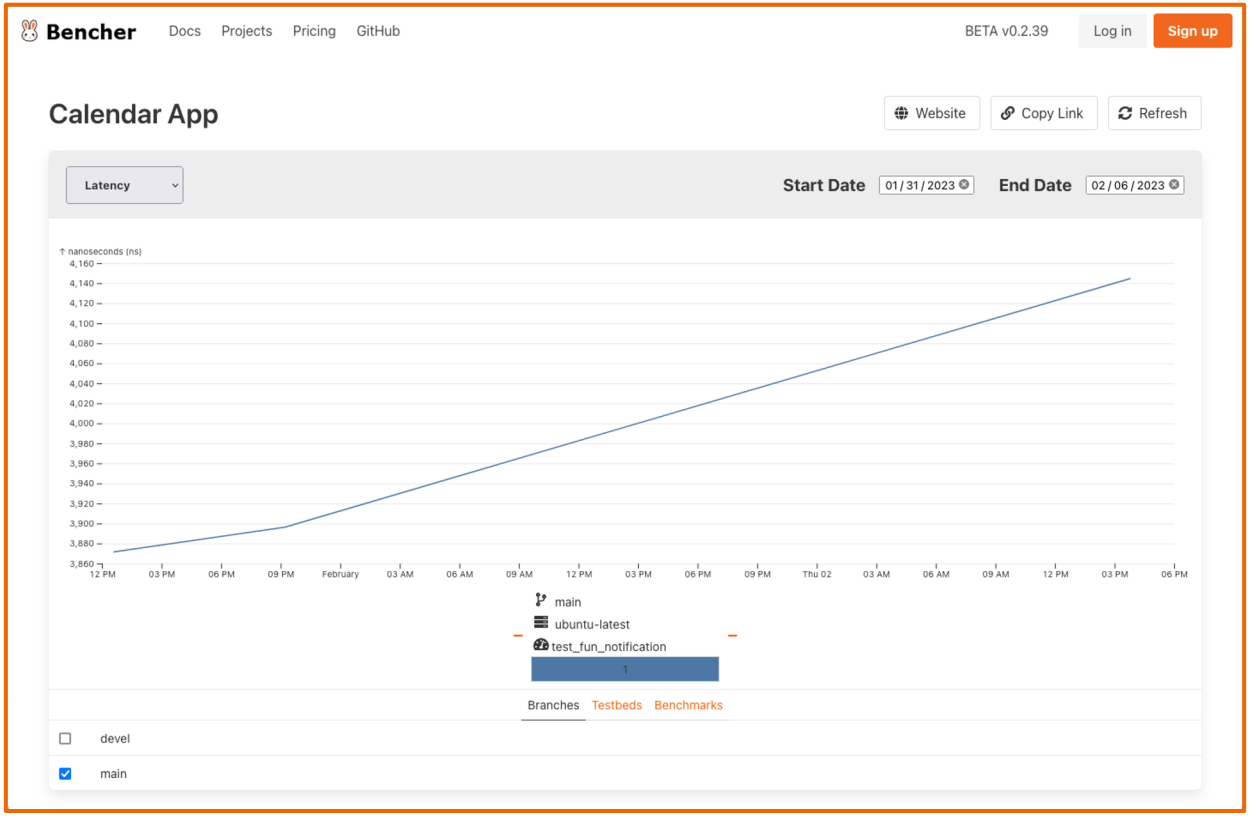
- Open Source
- Self-Hosted & SaaS
- Multitenancy
- Multi-Language
- Statistical Thresholds & Alerts

Bencher

- Open Source
- Self-Hosted & SaaS
- Multitenancy
- Multi-Language
- Statistical Thresholds & Alerts
- GitHub Action

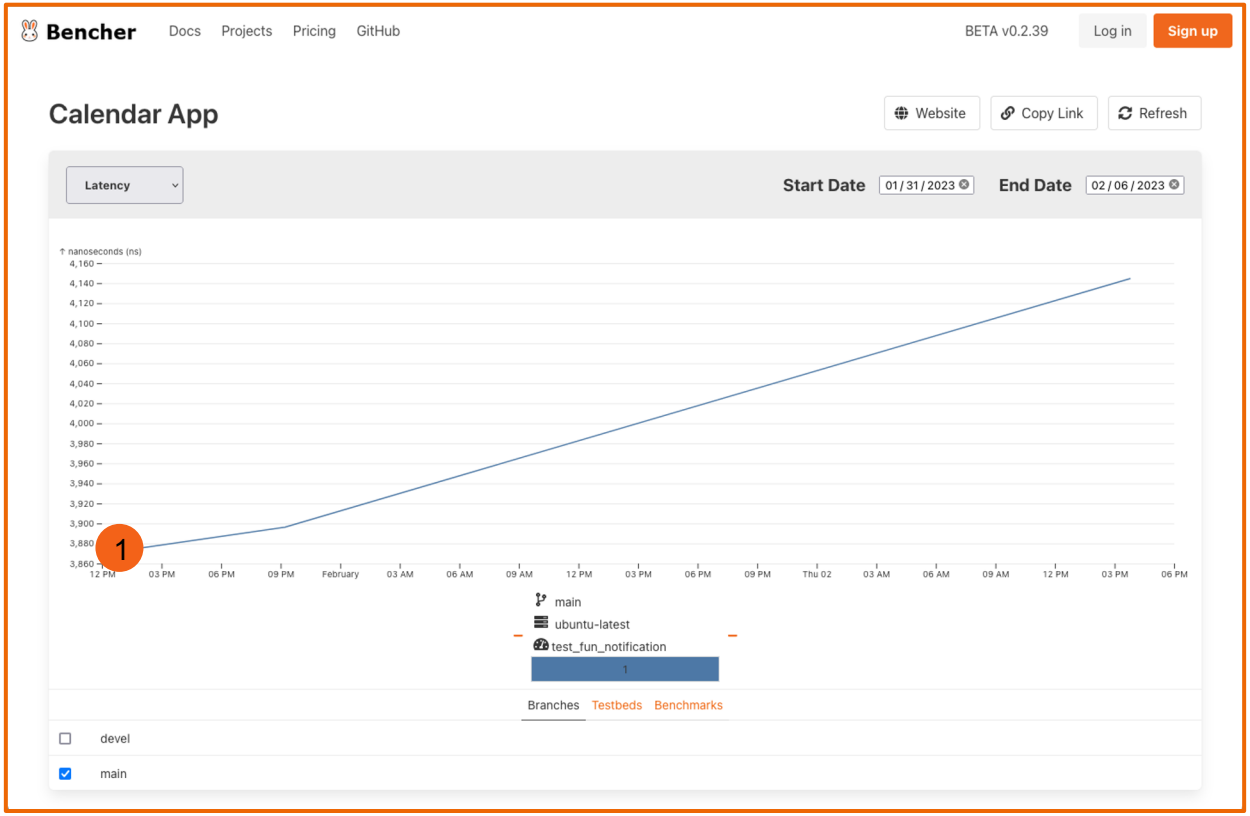
Track your benchmarks

Track Your Benchmarks

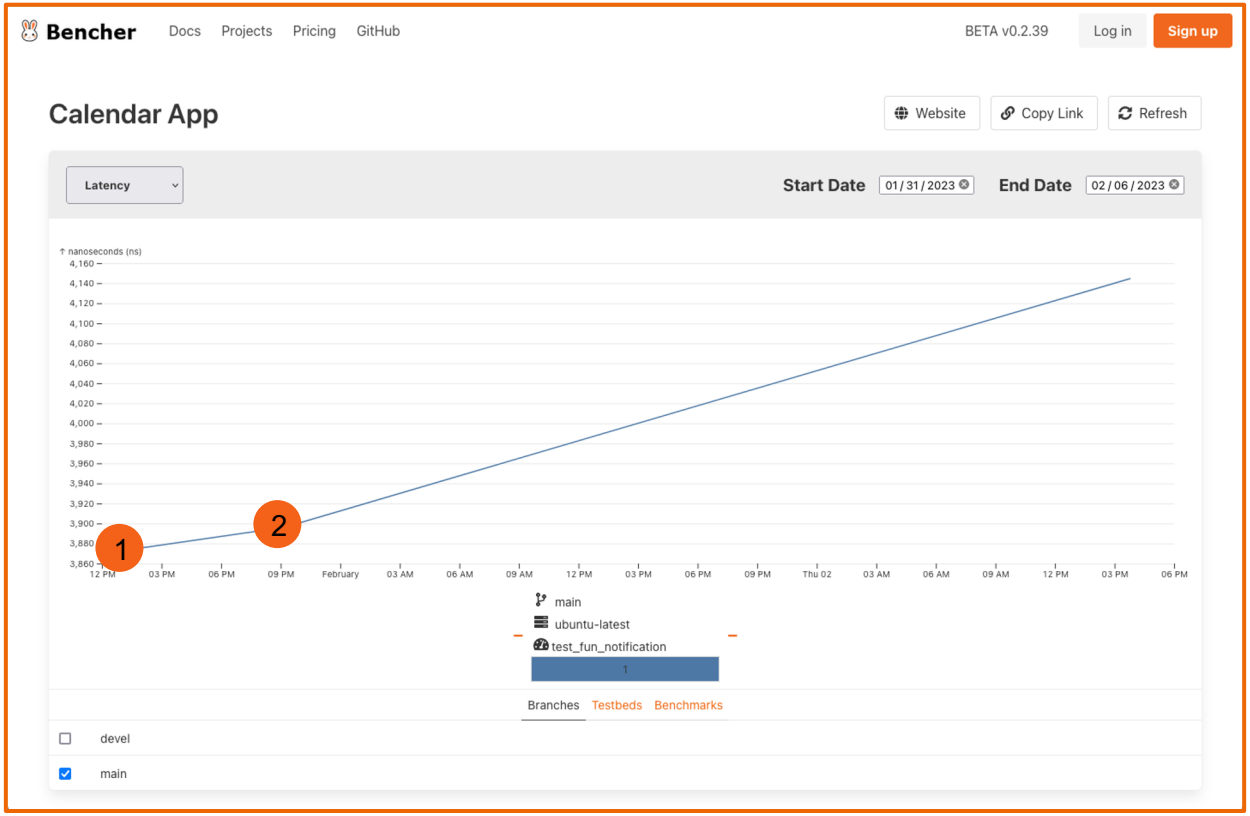


Bencher

Track Your Benchmarks

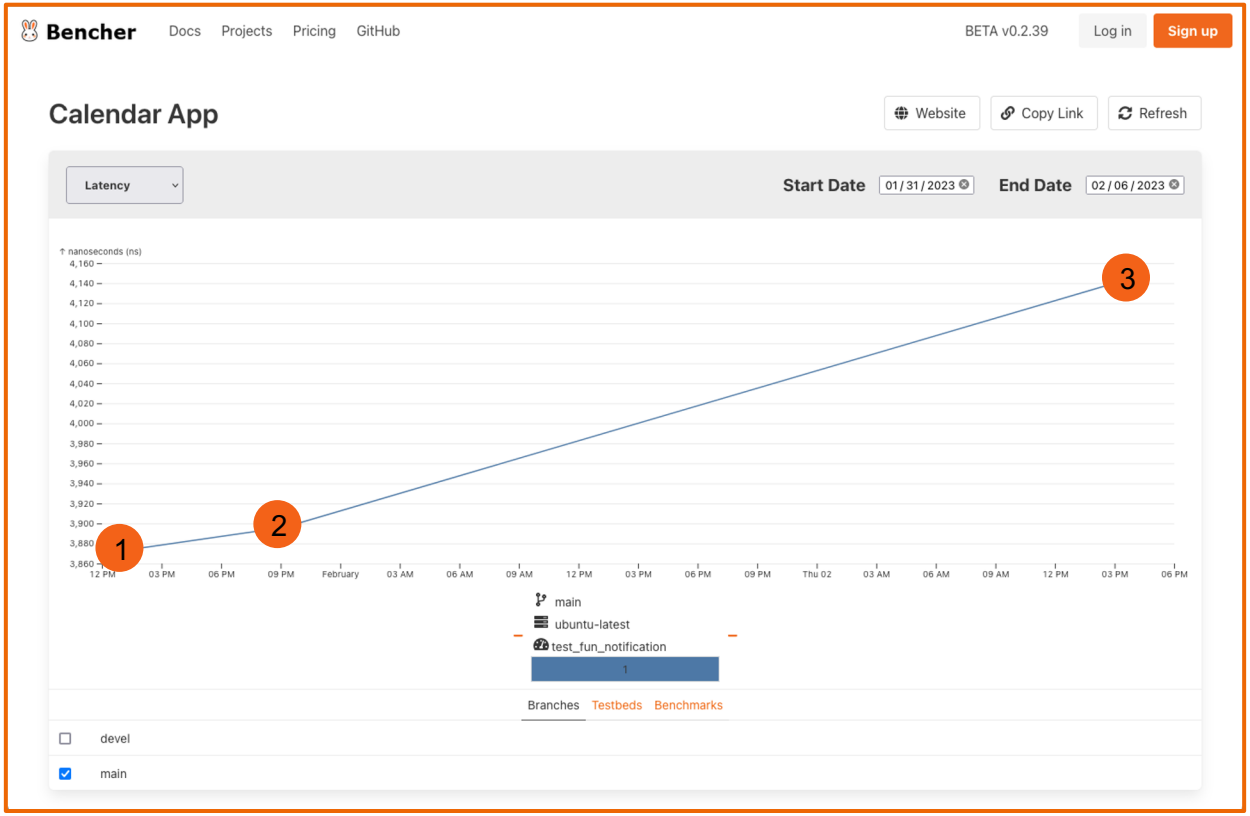


Track Your Benchmarks



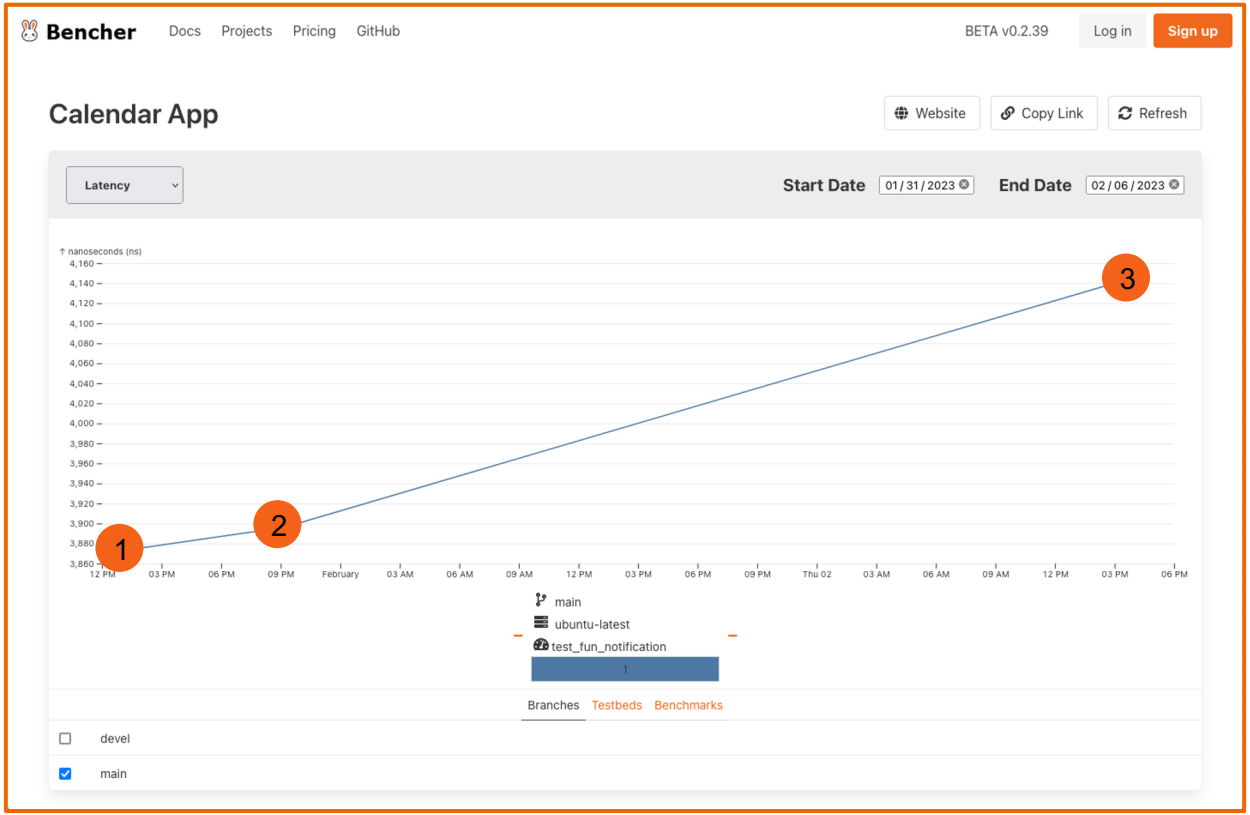
Bencher

Track Your Benchmarks



Bencher

Track Your Benchmarks



Bencher

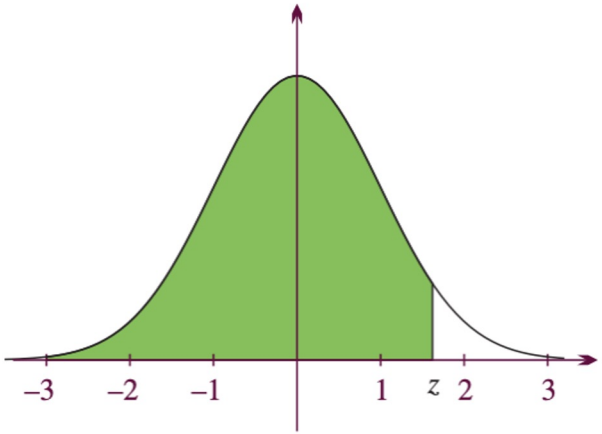
Detect performance regressions

Detect performance regressions

Detect performance regressions

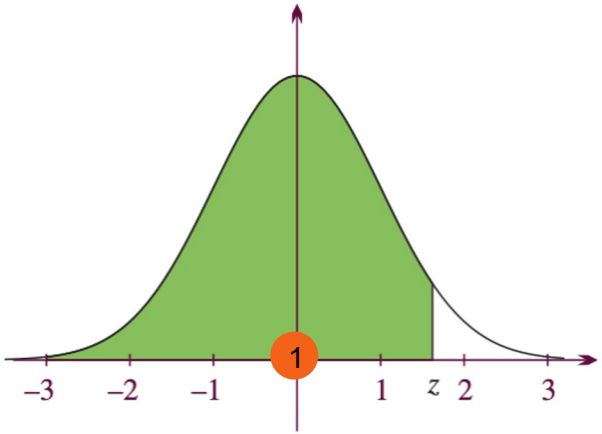
Bencher

Detect performance regressions



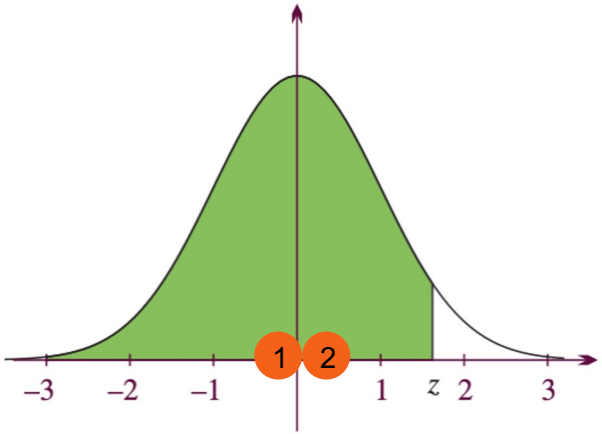
Bencher

Detect performance regressions



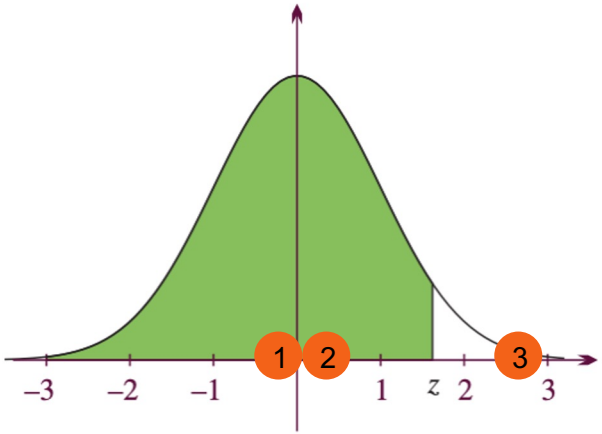
Bencher

Detect performance regressions



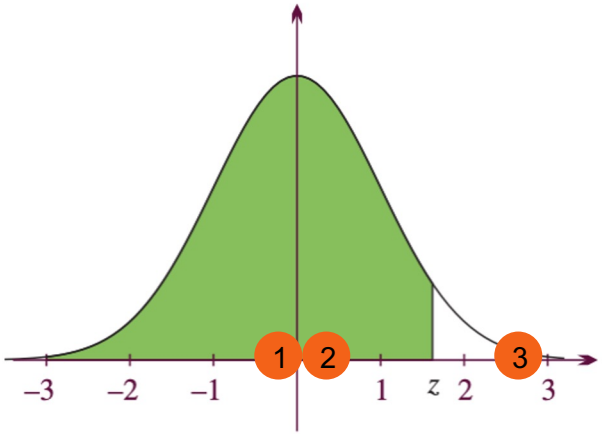
Bencher

Detect performance regressions



Bencher

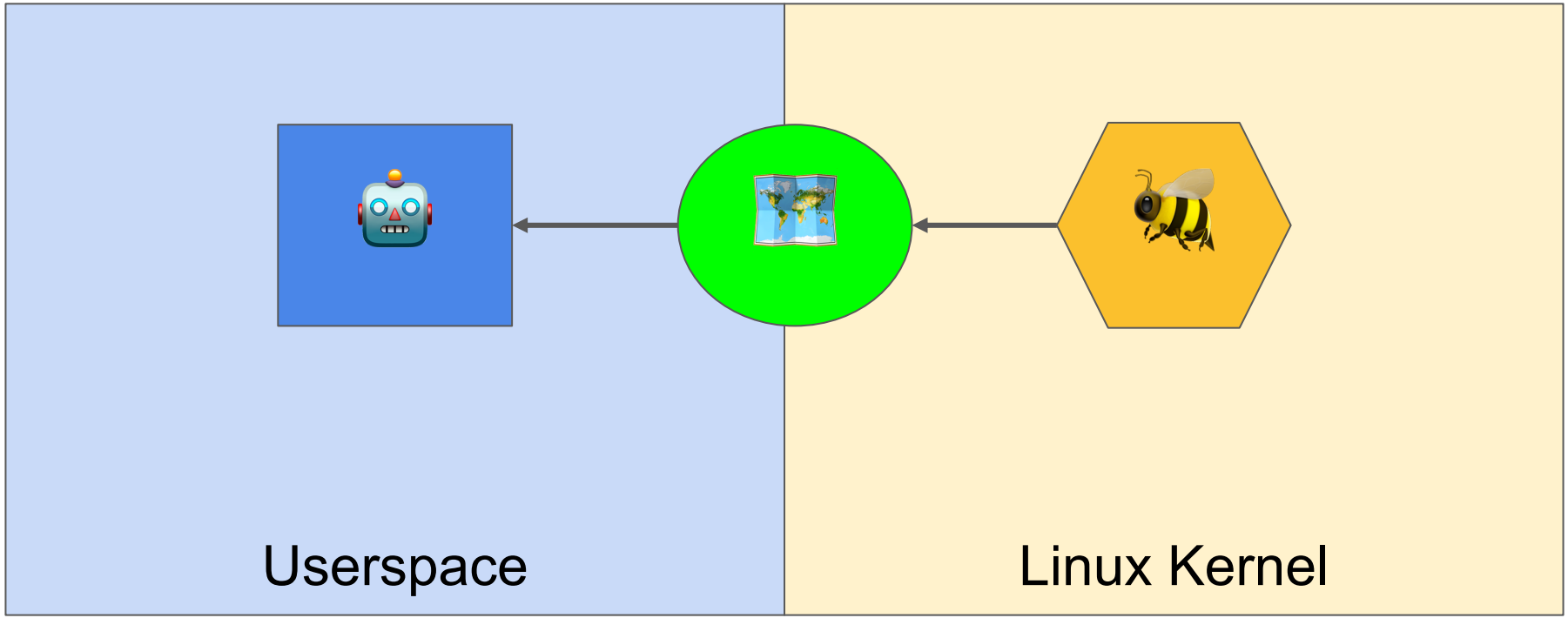
Detect performance regressions



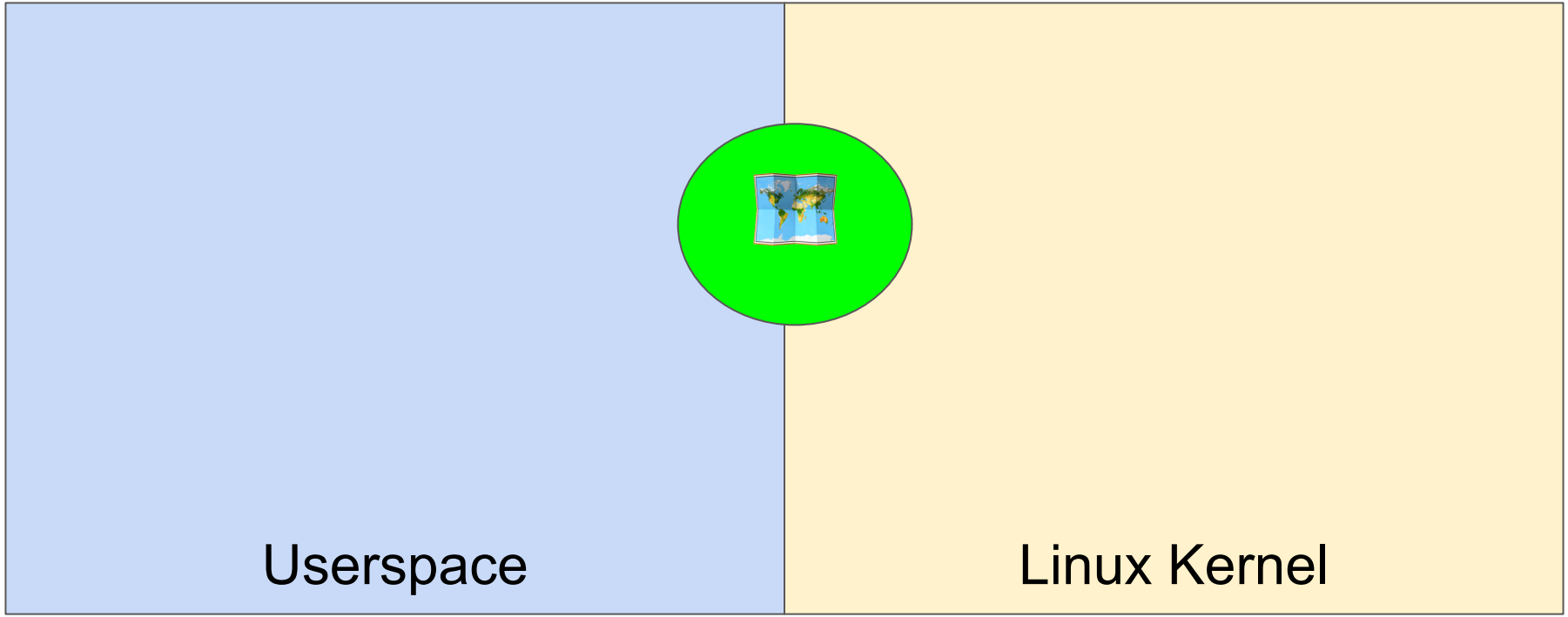
Bencher

Continuous Micro-Benchmarking with Criterion

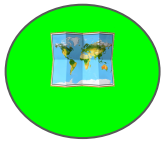
Continuous Micro-Benchmarking with Criterion



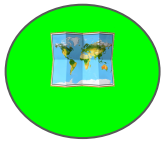
Continuous Micro-Benchmarking with Criterion



Continuous Micro-Benchmarking with Criterion

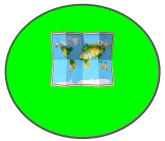


Continuous Micro-Benchmarking with Criterion



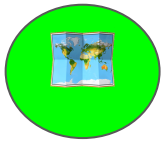
bencher run \

Continuous Micro-Benchmarking with Criterion



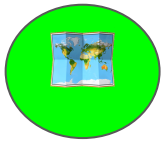
```
bencher run \
    --adapter rust_criterion \
```

Continuous Micro-Benchmarking with Criterion



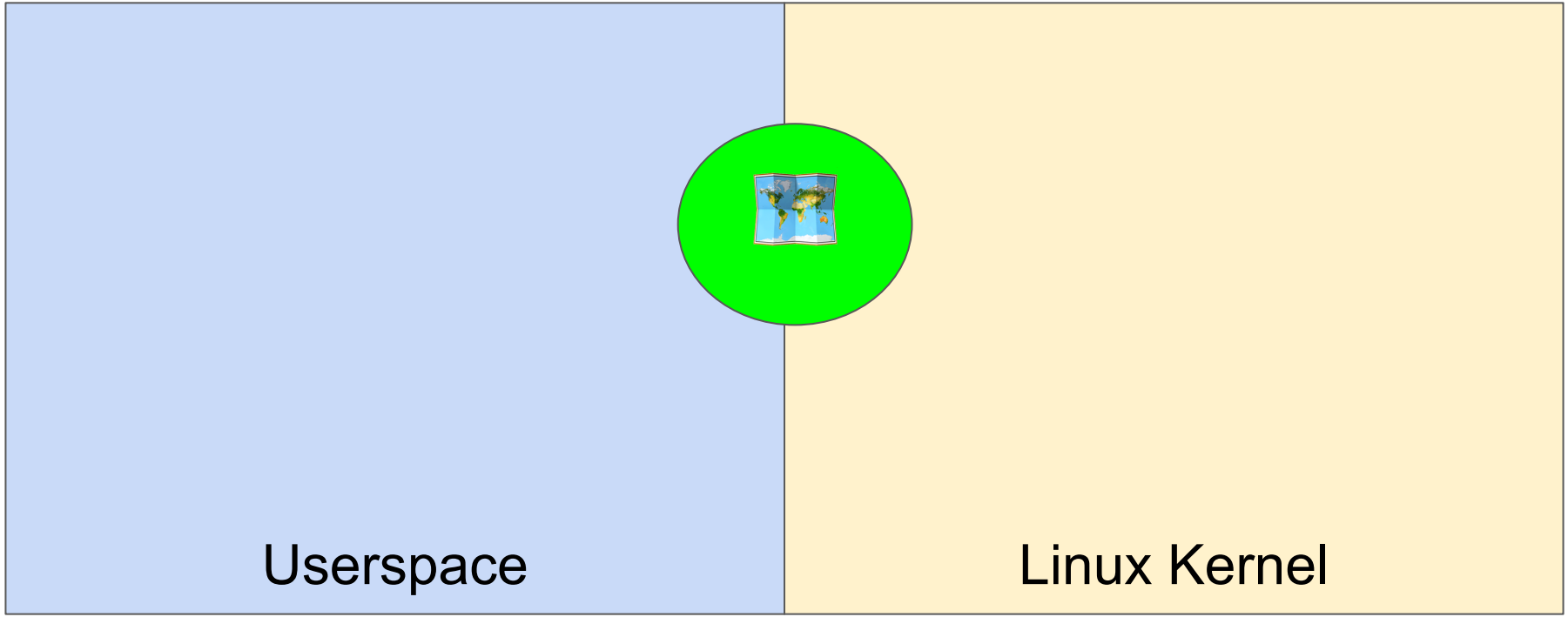
```
bencher run \
    --adapter rust_criterion \
    “cargo bench”
```

Continuous Micro-Benchmarking with Criterion

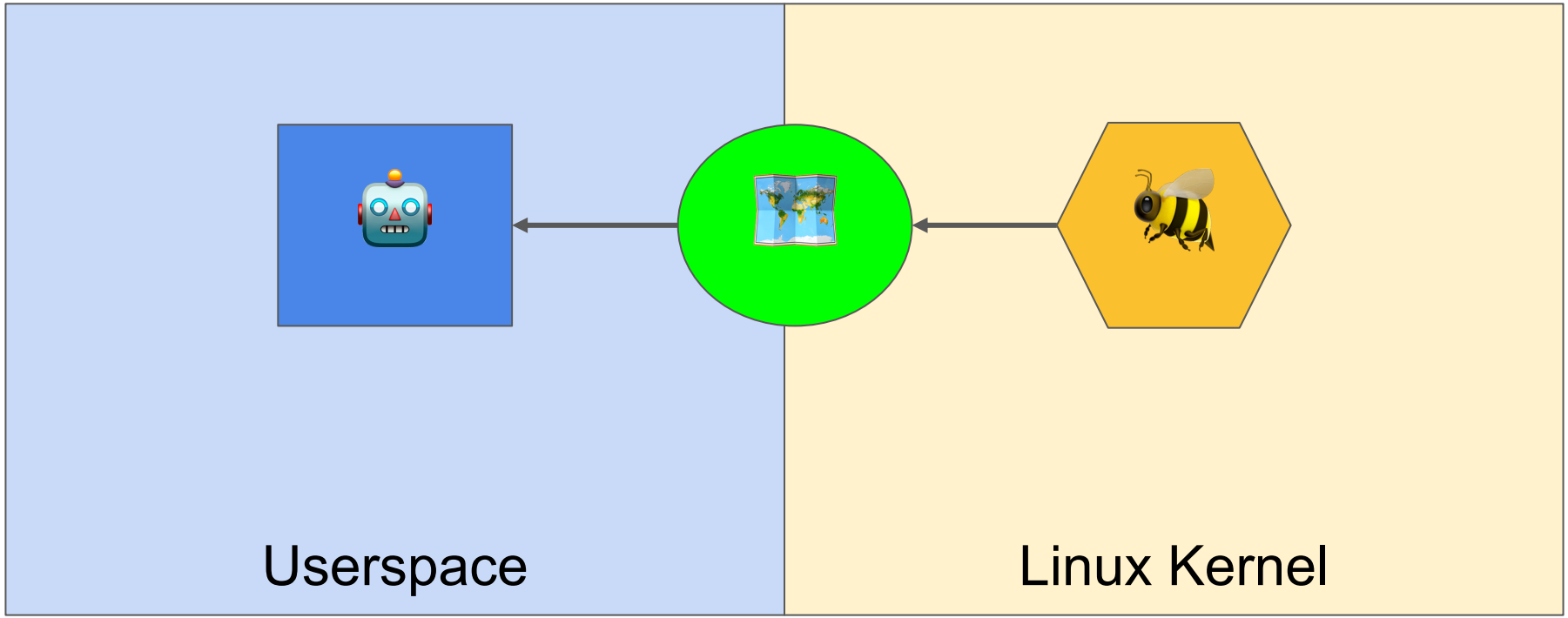


```
bencher run \
    --adapter rust_criterion \
    --err
    “cargo bench”
```

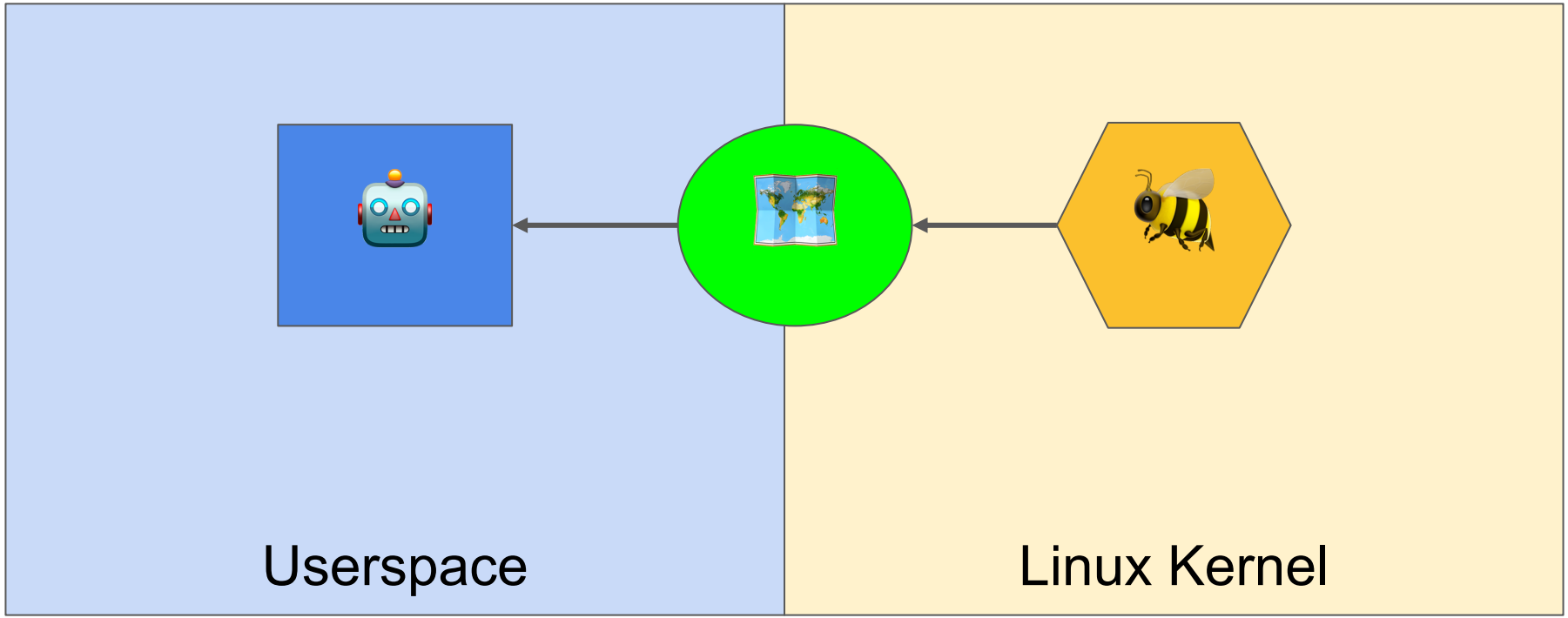
Continuous Micro-Benchmarking with Criterion



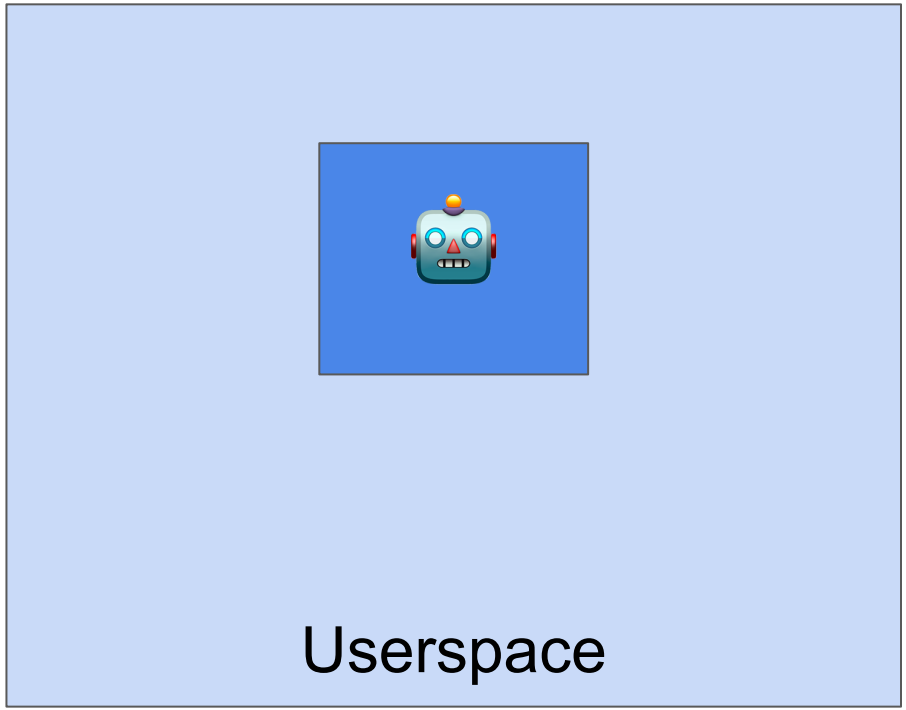
Continuous Micro-Benchmarking with Criterion



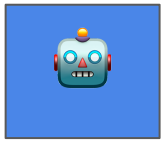
Continuous Macro-Benchmarking Custom Harness



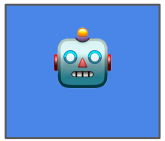
Continuous Macro-Benchmarking Custom Harness



Continuous Macro-Benchmarking Custom Harness

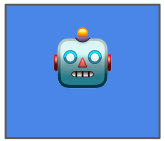


Continuous Macro-Benchmarking Custom Harness



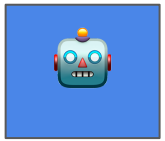
```
bencher run \
```

Continuous Macro-Benchmarking Custom Harness



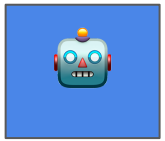
```
bencher run \
    --adapter json \
```

Continuous Macro-Benchmarking Custom Harness



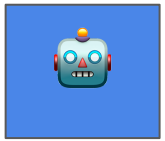
```
bencher run \
    --adapter json \
    --file ../target/results.json \
```

Continuous Macro-Benchmarking Custom Harness



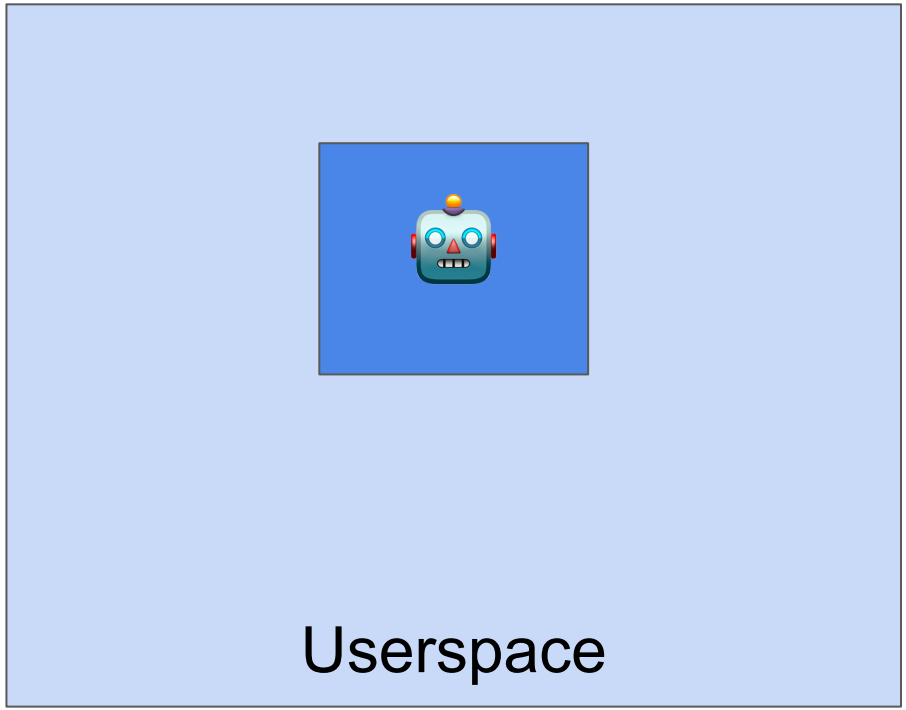
```
bencher run \
    --adapter json \
    --file ../target/results.json \
    “sudo -E $(which cargo) bench”
```

Continuous Macro-Benchmarking Custom Harness

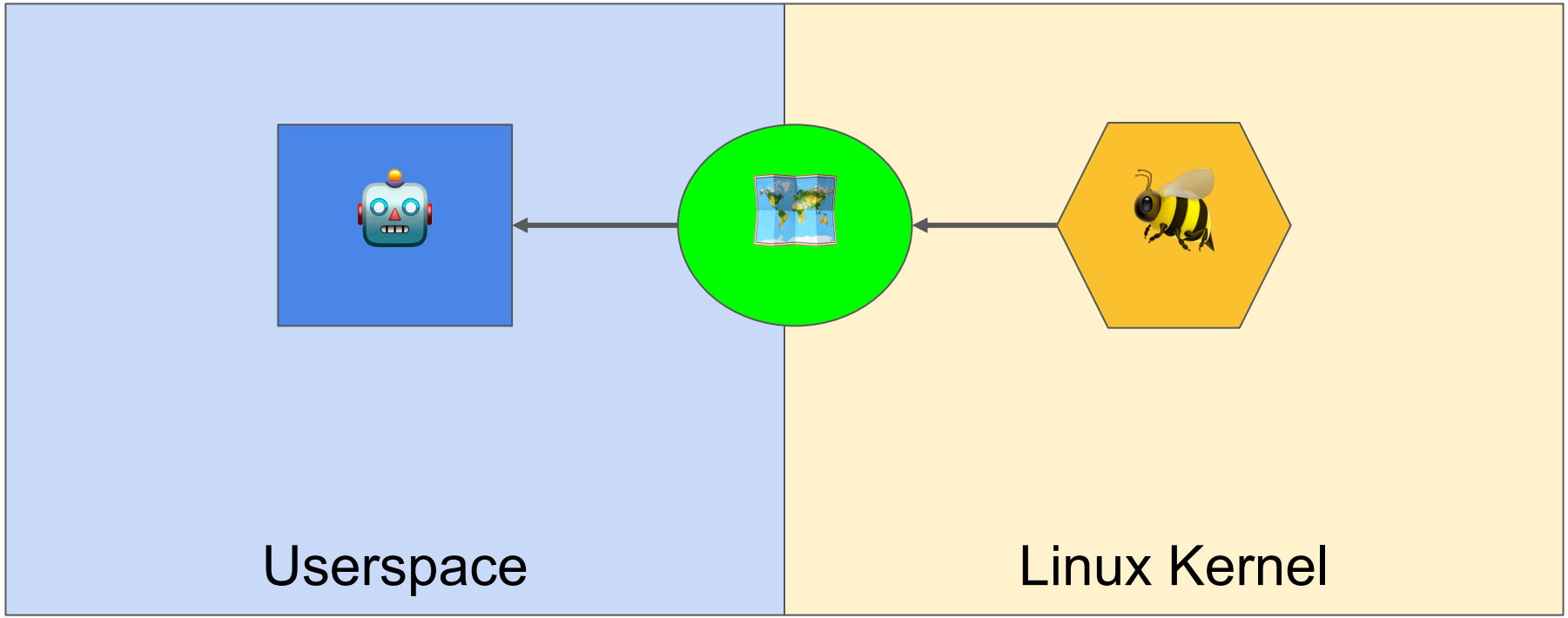


```
bencher run \
    --adapter json \
    --file ../target/results.json \
    --err
    “sudo -E $(which cargo) bench”
```

Continuous Macro-Benchmarking Custom Harness



Continuous Macro-Benchmarking Custom Harness



Catch performance regressions in CI

Add Bencher to GitHub Actions

```yaml
benchmark:
  name: Run benchmarks
  runs-on: ubuntu-latest
  env:
    BENCHER_PROJECT: fun_xdp
    BENCHER_TESTBED: ubuntu-latest
    BENCHER_ADAPTER: rust_criterion
    BENCHER_API_TOKEN: ${{ secrets.BENCHER_API_TOKEN }}
  steps:
    - uses: actions/checkout@v3
    - uses: bencherdev/bencher@v0.3.15
    - name: Benchmark with Bencher
      run: |
        cd examples/ebpf/ebp-common
        bencher run --if-branch $GITHUB_REF_NAME --err "cargo bench"
```

Bencher

Continuous Benchmarking

Continuous Benchmarking

- Detection ➡️ Prevention

Continuous Benchmarking

- Detection ➡️ Prevention
- Production is too late

Continuous Benchmarking

- Detection ➡️ Prevention
- Production is too late
- Development is local only

Continuous Benchmarking

- Detection ➡️ Prevention
- Production is too late
- Development is local only
- Continuous Benchmarking can save us a lot of pain

Continuous Benchmarking

- Detection ➡️ Prevention
- Production is too late
- Development is local only
- Continuous Benchmarking can save us a lot of pain
- Don’t reinvent the wheel, again

Bencher High Dimensionality

Bencher High Dimensionality

- Branch (ex `main`)

Bencher High Dimensionality

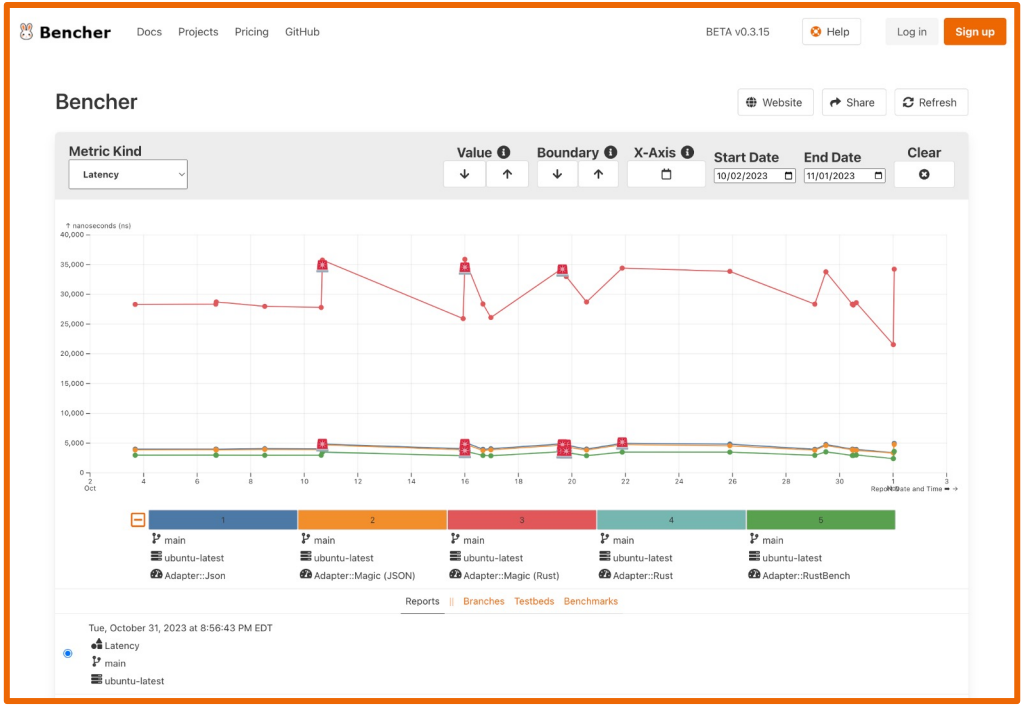
- Branch (ex `main`)
- Testbed (ex `ubuntu-latest`)

Bencher High Dimensionality

- Branch (ex `main`)
- Testbed (ex `ubuntu-latest`)
- Benchmark (ex `fun_xdp`)

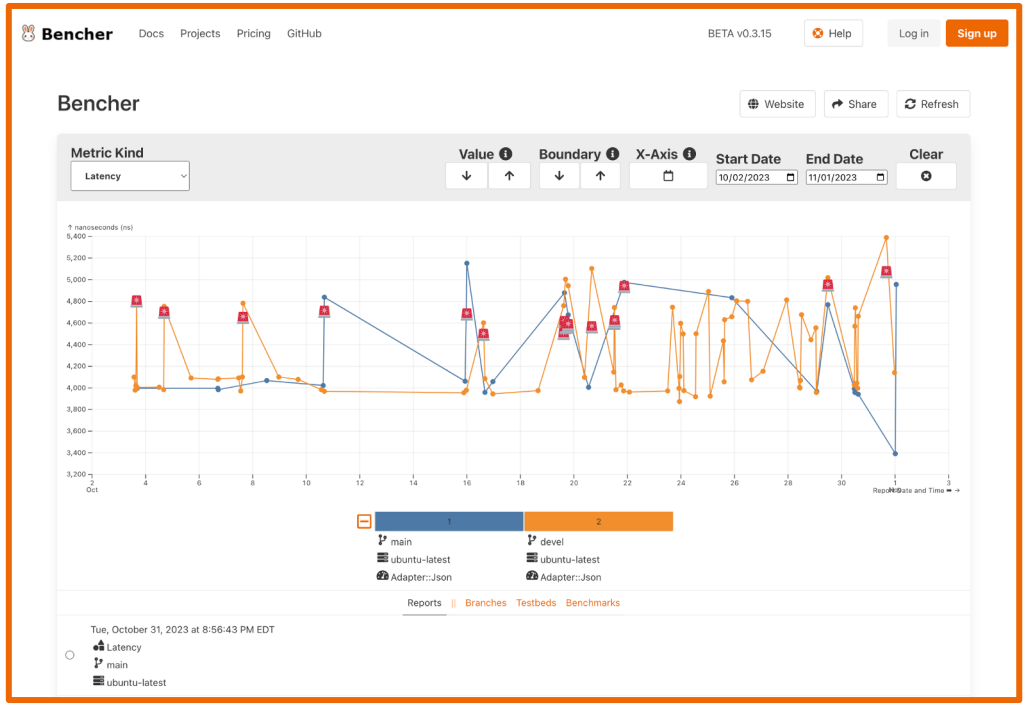
Bencher High Dimensionality

- Branch (ex `main`)
- Testbed (ex `ubuntu-latest`)
- Benchmark (ex `fun_xdp`)
- Metric Kind (ex `latency`)

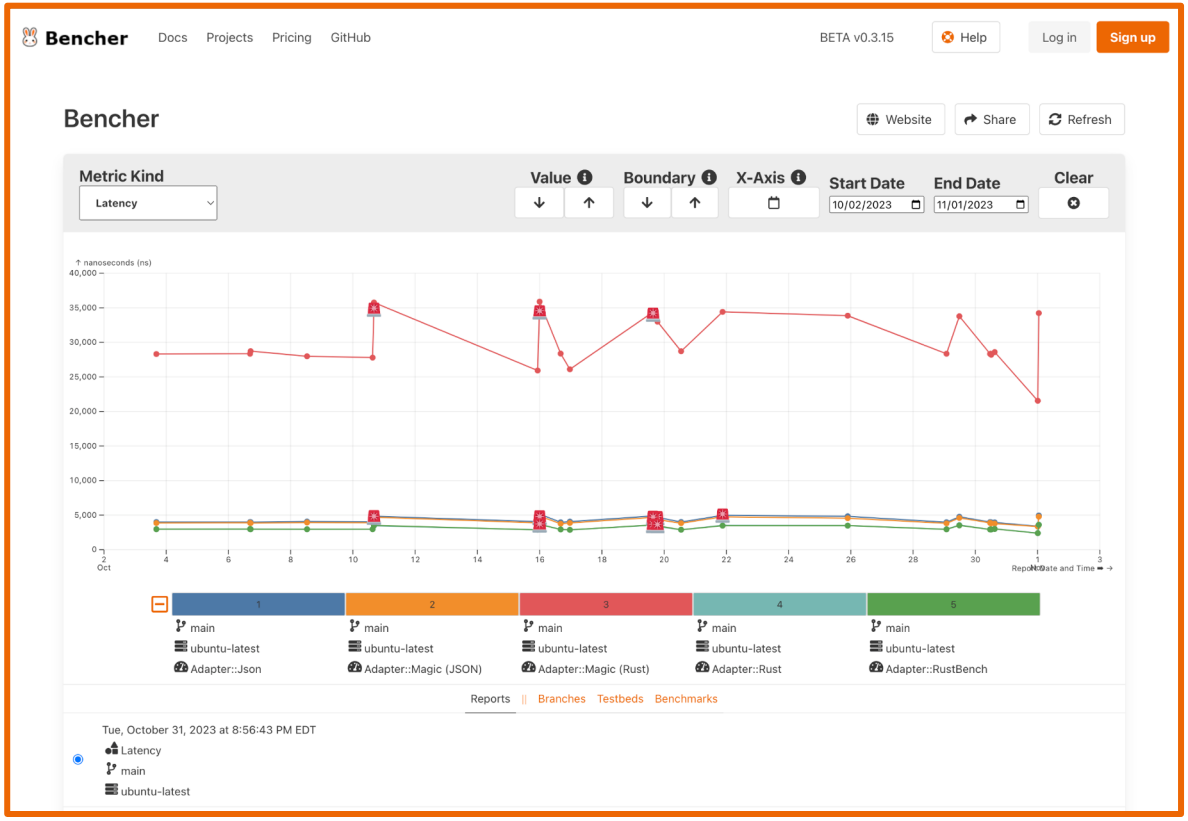


Bencher High Dimensionality

- Branch (ex `main`)
- Testbed (ex `ubuntu-latest`)
- Benchmark (ex `fun_xdp`)
- Metric Kind (ex `latency`)



Public Perf Page

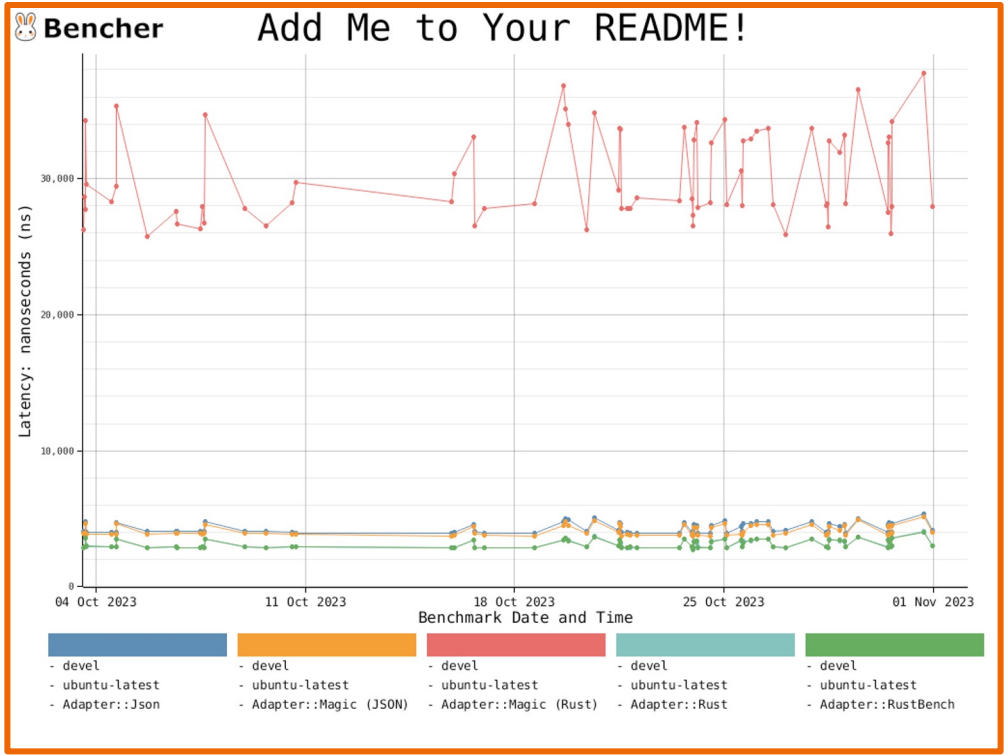


Bencher

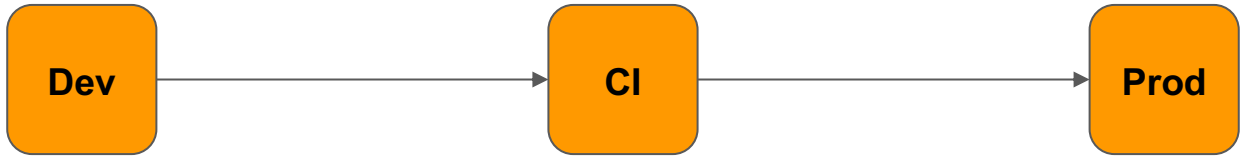
Embedded Perf Plots

Bencher

Auto-Updated Perf Plot Images



When do performance regression get detected?



❌ Local Only

✅ Continuous Benchmarking

❌ Too Late

Overview

Overview

- Basic eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust
- Benchmark eBPF program in Rust

Overview

- Basic eBPF program in Rust
- Evolve eBPF program in Rust
- Benchmark eBPF program in Rust
- Continuous Benchmarking eBPF program in Rust

Run Fast!

Catch Performance Regressions in eBPF with Rust

https://github.com/bencherdev/bencher

Run Fast!

Catch Performance Regressions in eBPF with Rust

https://github.com/bencherdev/bencher

bencher.dev/repo

Run Fast!

Catch Performance Regressions in eBPF with Rust

https://github.com/bencherdev/bencher

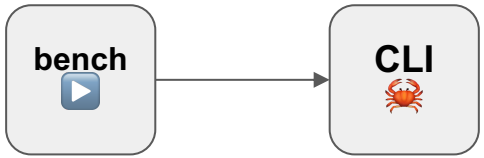
bencher.dev/repo

Bencher

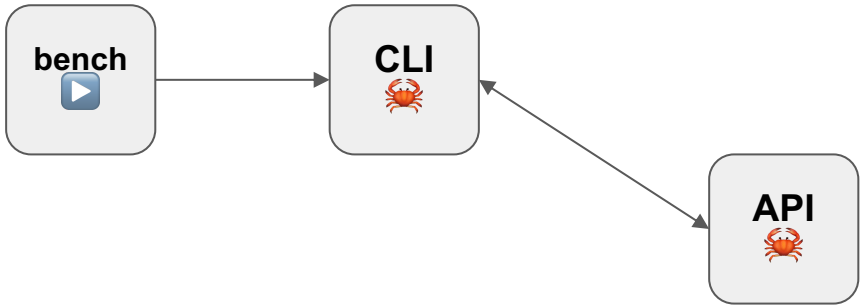
Bencher Architecture

Bencher Architecture

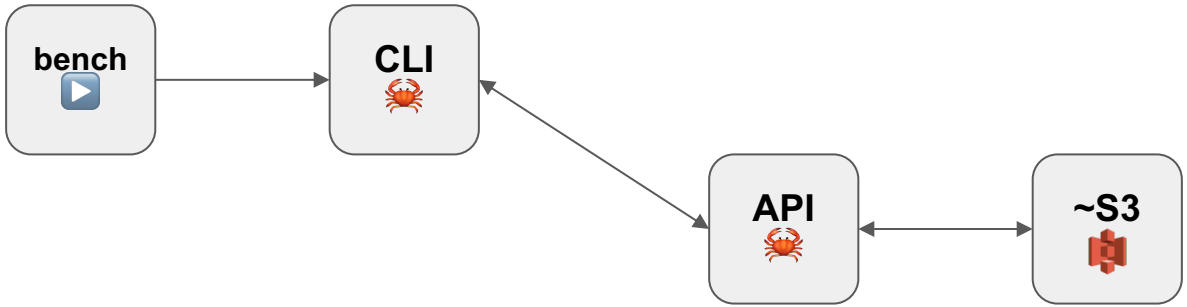
Bencher Architecture



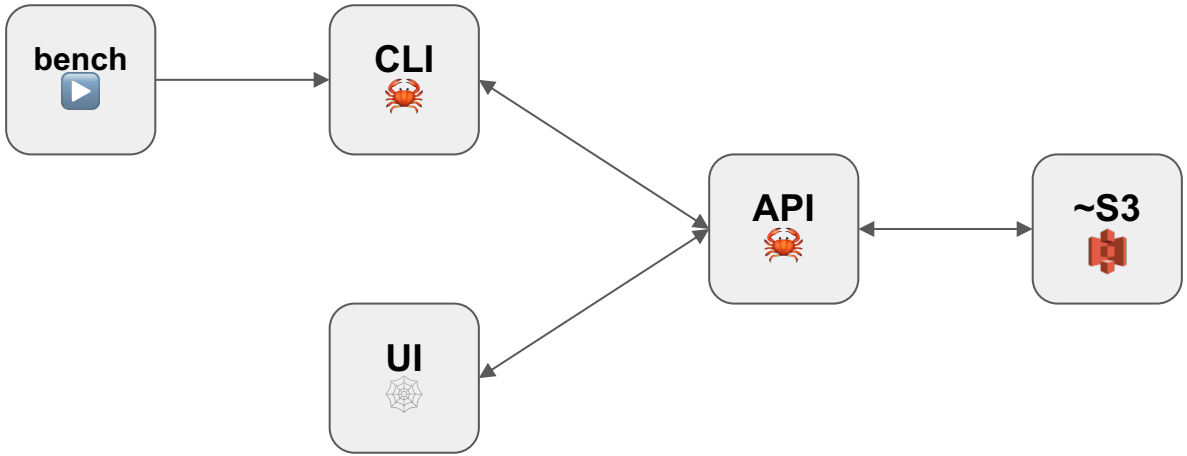
Bencher Architecture



Bencher Architecture



Bencher Architecture



Enable kernel.bpf_stats_enabled

Enable kernel.bpf_stats_enabled: sysctl

5.1

Enable kernel.bpf_stats_enabled: sysctl

```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
```

Enable kernel.bpf_stats_enabled: sysctl

```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
```

Enable kernel.bpf_stats_enabled: sysctl

```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
$ cat /proc/sys/kernel/bpf_stats_enabled
```

Enable kernel.bpf_stats_enabled: sysctl

```
$ sudo sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
$ cat /proc/sys/kernel/bpf_stats_enabled
1
```

Enable kernel.bpf_stats_enabled: syscall

5.8

Enable kernel.bpf_stats_enabled: syscall

```rust
fn bpf_prog_enable_stats(prog_fd: RawFd) -> Result<bpf_prog_info, io::Error> {
    let mut attr = unsafe { mem::zeroed::<bpf_attr>() };
    // info gets entirely populated by the kernel
    let info = MaybeUninit::zeroed();

    attr.enable_stats.type = BPF_STATS_RUN_TIME;

    match sys_bpf(bpf_cmd::BPF_ENABLE_STATS, &attr, mem::size_of::<bpf_attr>()) {
        Ok(_) => Ok(unsafe { info.assume_init() }),
        Err((_, err)) => Err(err),
    }
}
```

View kernel.bpf_stats_enabled

View kernel.bpf_stats_enabled: bpftool

5.1

View kernel.bpf_stats_enabled: bpftool

5.1

```
$ sudo bpftool prog show
```

View kernel.bpf_stats_enabled: bpftool

5.1

```
$ sudo bpftool prog show
...
241: xdp  name fun_xdp  tag 2a12b6a683c5a7b6  gpl run_time_ns 9682091 run_cnt 1043
        loaded_at 2023-04-30T09:09:33-0400  uid 0
        xlated 5600B  jited 5200B  memlock 8192B  map_ids 247,244,246,245
```

Bencher

View kernel.bpf_stats_enabled: /proc/<pid>/fdinfo/<fd>

5.1

View kernel.bpf_stats_enabled: /proc/<pid>/fdinfo/<fd>

5.1

```
$ sudo cat /proc/245216/fdinfo/17
```

View kernel.bpf_stats_enabled: /proc/<pid>/fdinfo/<fd>

5.1

```
$ sudo cat /proc/245216/fdinfo/17
pos:    0
flags:  02000002
mnt_id: 14
prog_type:      6
prog_jited:     1
prog_tag:       2a12b6a683c5a7b6
memlock:        8192
prog_id:        241
run_time_ns:    260673612
run_cnt:        28980
```

View kernel.bpf_stats_enabled: syscall

4.13

View kernel.bpf_stats_enabled: syscall

4.13

```rust
pub(crate) fn bpf_prog_get_info_by_fd(prog_fd: RawFd) -> Result<bpf_prog_info, io::Error> {
    let mut attr = unsafe { mem::zeroed::<bpf_attr>() };
    // info gets entirely populated by the kernel
    let info = MaybeUninit::zeroed();

    attr.info.bpf_fd = prog_fd as u32;
    attr.info.info = &info as *const _ as u64;
    attr.info.info_len = mem::size_of::<bpf_prog_info>() as u32;

    match sys_bpf(bpf_cmd::BPF_OBJ_GET_INFO_BY_FD, &attr) {
        Ok(_) => Ok(unsafe { info.assume_init() }),
        Err((_, err)) => Err(err),
    }
}
```