MYRMIC
Run anywhere

WebAssembly on Constrained Devices

Runtimes, tooling, and real-world trade-offs in Rust.

Fedor Smirnov, Ph.D.

FOSDEM 26, 01.02.26

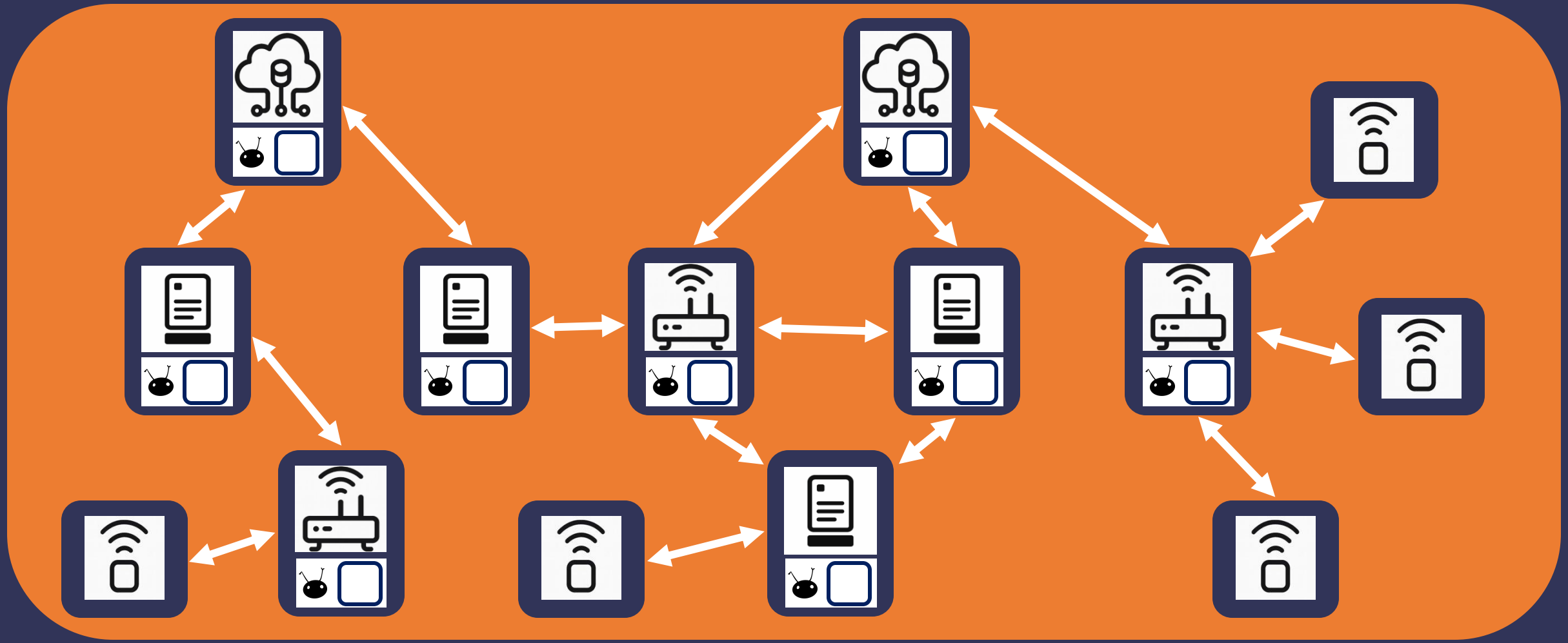
Outline

- Motivation and context
- Module considerations
- Wasm runtime trade-offs and integration
- Conclusion

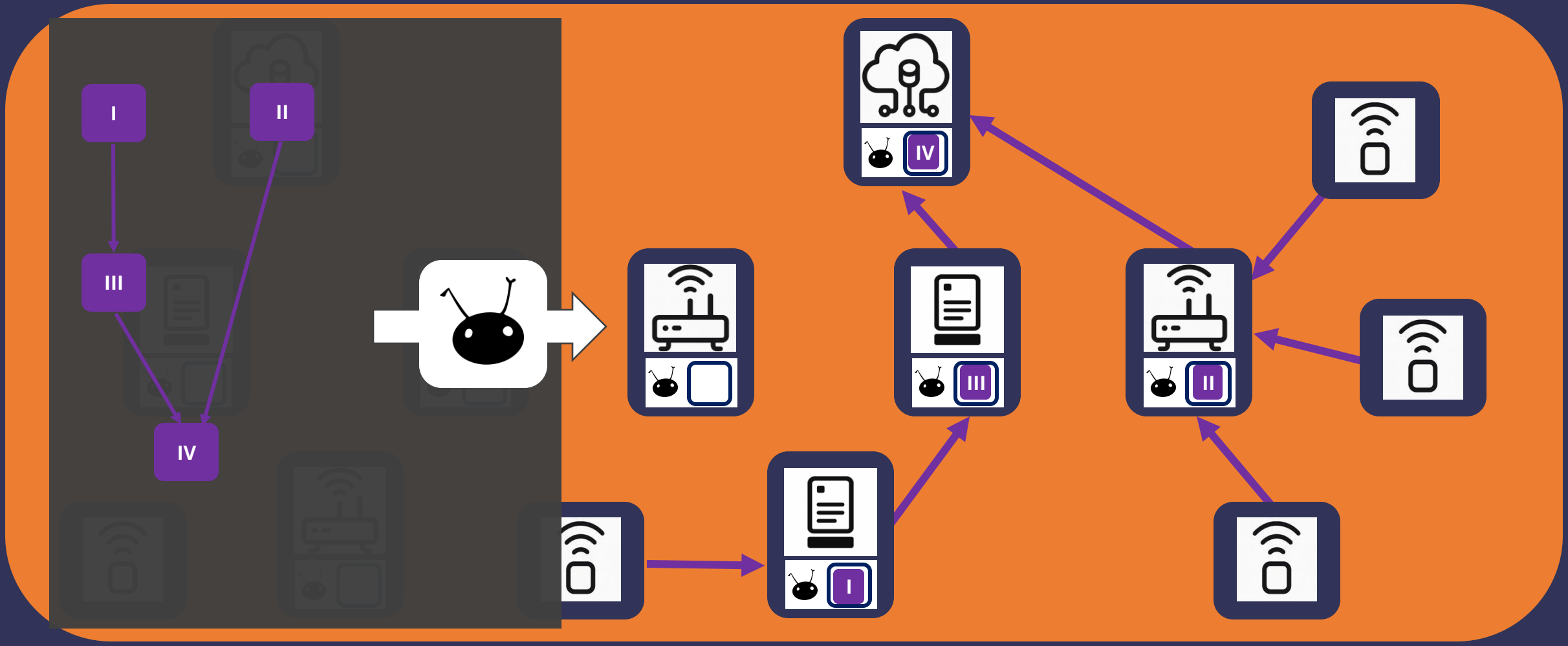
Motivation

Heterogeneous Distributed Systems

A Common Middleware Layer



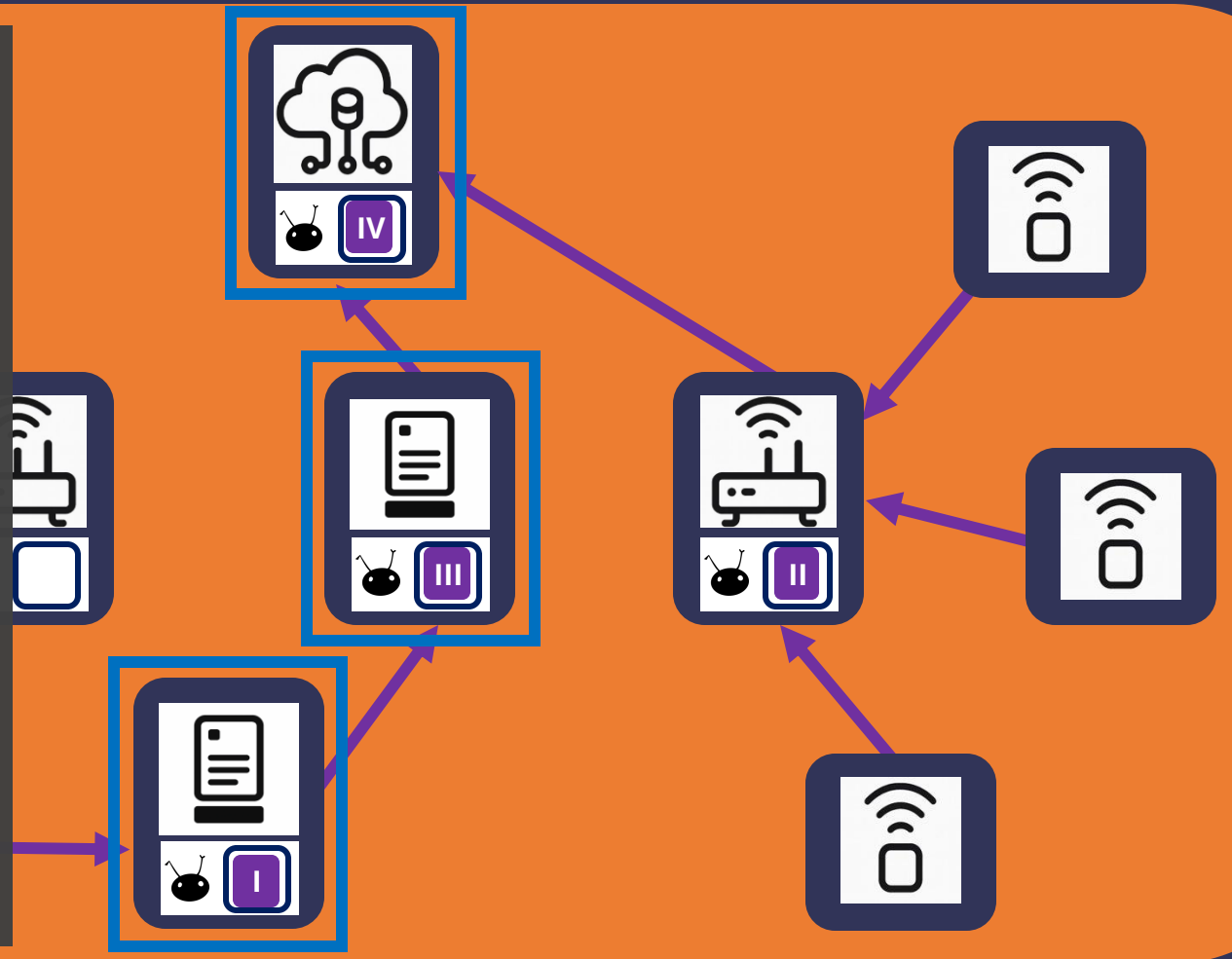
Running Application Code Across Devices



Wasm on OS-Based Devices

On OS-grade devices, Wasm is the natural choice due to:

- Strong sandboxing
- Target-independent deployment
- Mature tooling and ecosystem



Wasm on Microcontrollers

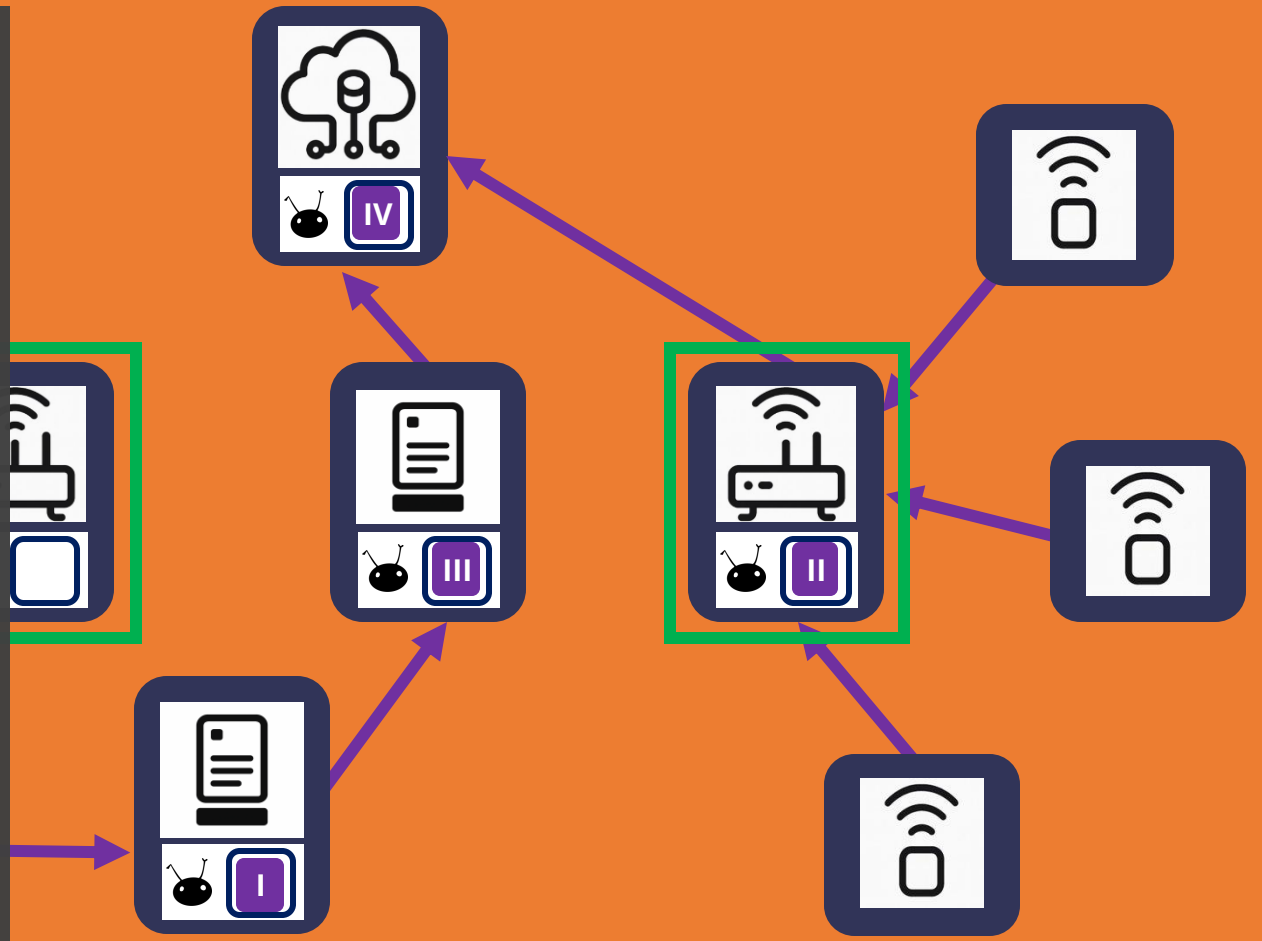
On Microcontrollers, choosing Wasm is far less obvious.

What we gain:

- Dynamic updates of device behavior without reflashing
- Isolation of application logic from firmware
- Reduced risk of "bricking"

What it costs

- Additional memory
- Performance overhead



Alternatives for Deploying Code on MCUs

Approach	Dynamic Updates	Safety/Isolation	Footprint	Performance	Tooling/ Ecosystem
Native firmware (Rust / C)	❌	❌	✅	✅	✅
Static plugins / configuration	⚠️	❌	✅	✅	⚠️
Scripting languages (e.g. Lua)	✅	⚠️	❌	❌	⚠️
Custom VM/DSL (e.g. PLCs)	⚠️	⚠️	✅	⚠️	❌
WebAssembly	✅	✅	⚠️	⚠️	✅

Wasm on MCUs:
Practical Consequences

What Does Using Wasm Mean in Practice?

*Assume that you **decide to use WebAssembly on an MCU**.*
*What does this mean for **how you write and compile code**?*

- Compilation target and runtime assumptions
- Module <> host interaction
- Developer ergonomics

WASI vs `-unknown` on Microcontrollers

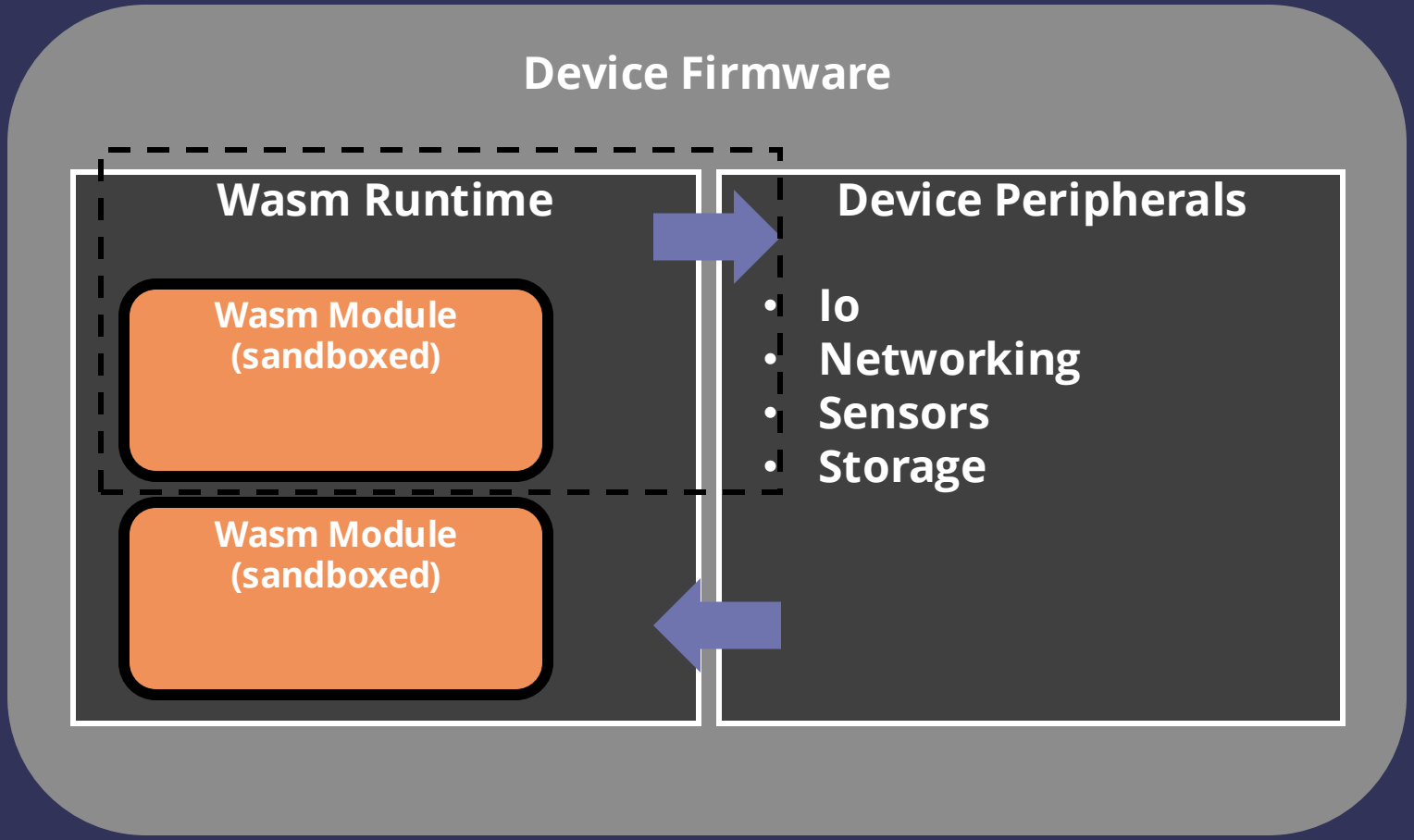
Why not `wasm32-wasip1`?

- Not (fully) supported by all embedded runtimes
- Requires a WASI layer in the runtime, increasing the runtime footprint
- Assumes OS-like environment (I/O, time, randomness), which is often non-trivial on bare metal

What `wasm32-unknown-unknown` implies

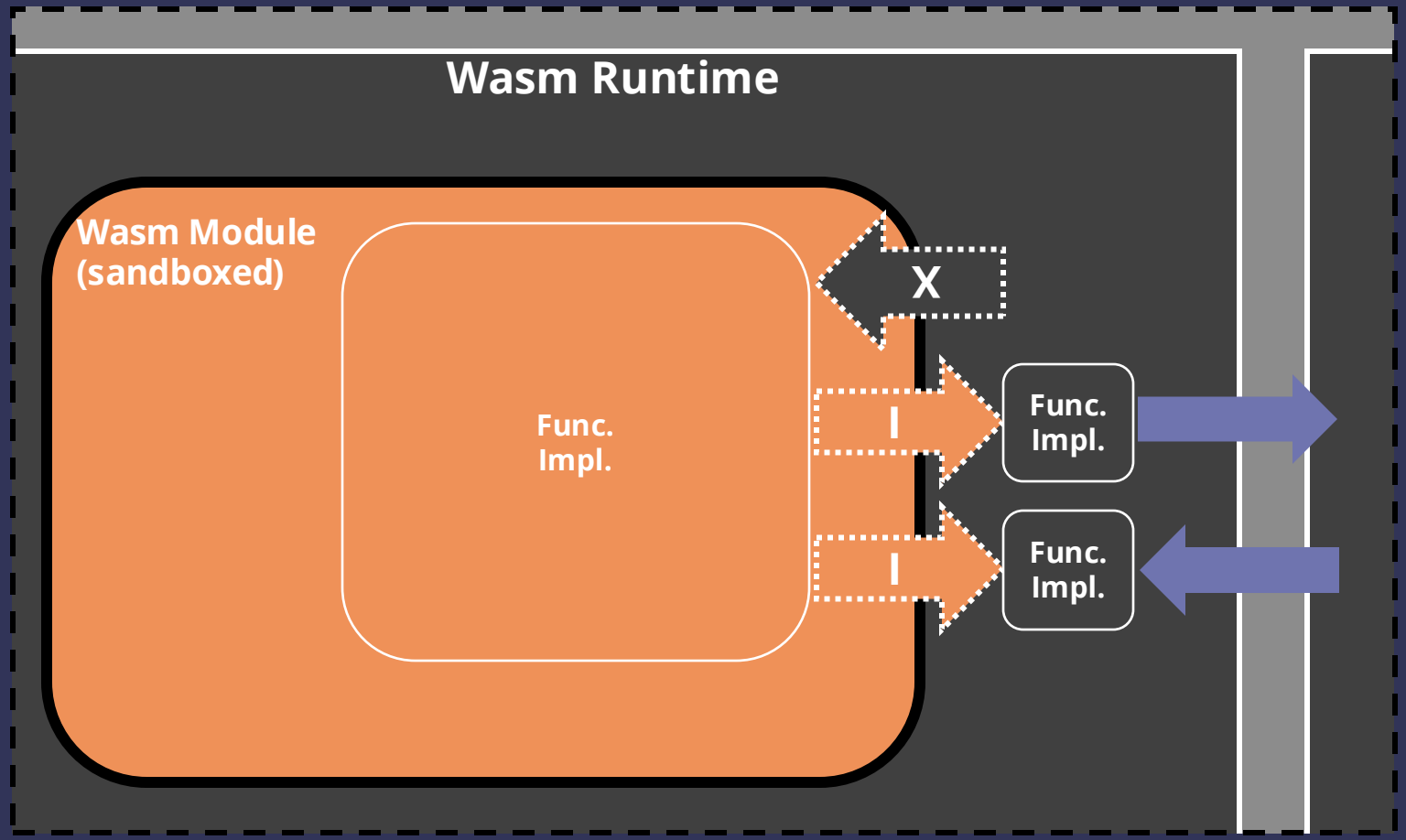
- Modules are no_std
- All functionality must be provided via host functions
- Allocation is explicit

Embedding Wasm into the Firmware Stack



- **Only the Wasm runtime (the *host*)** can interact with device peripherals
- Wasm modules (the *guests*) run inside isolated sandboxes and **cannot access hardware** (or the operating system)

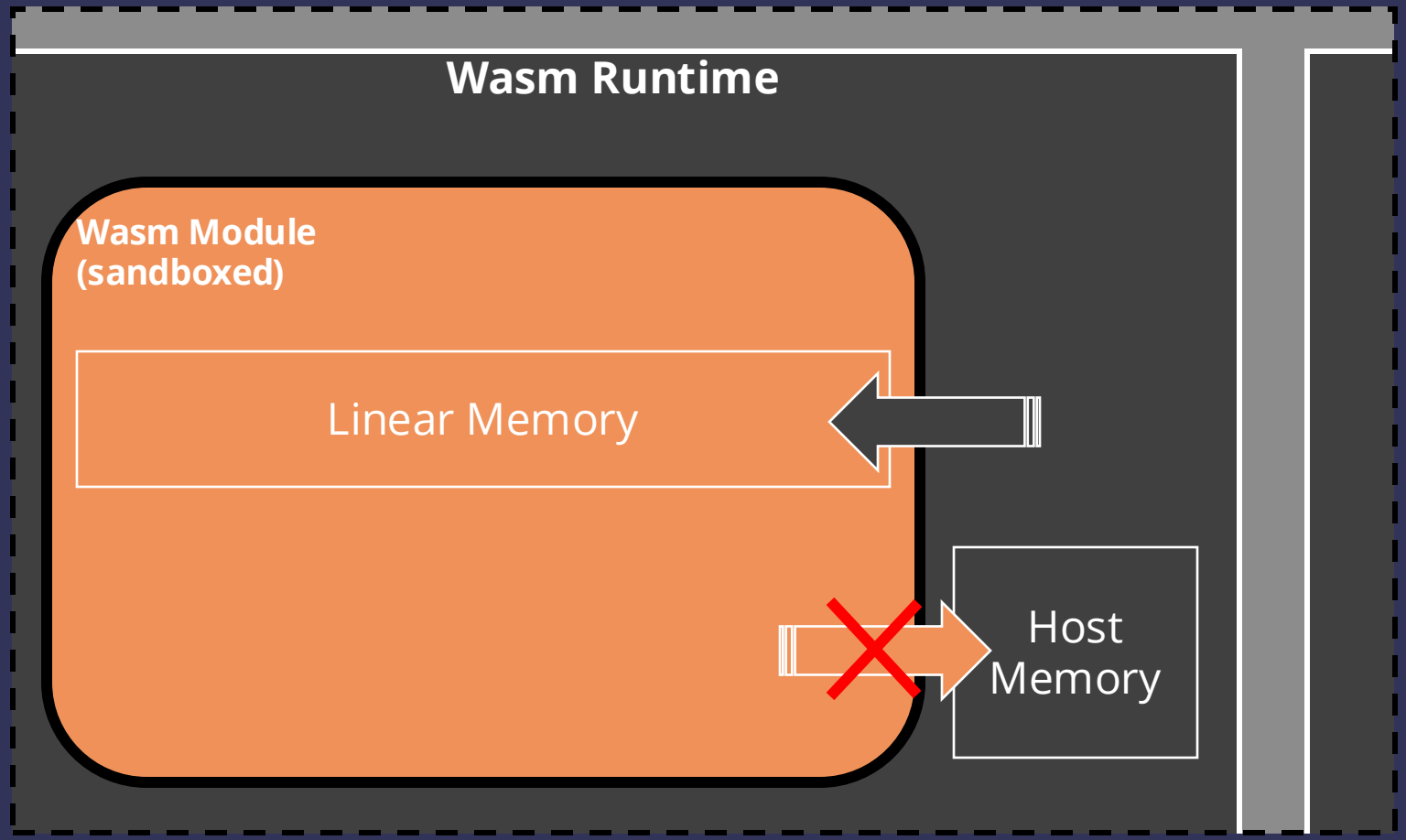
Host/Guest Interaction – Imports/Exports



- Host-guest interaction occurs **exclusively** through the module's **imports** (**I**) and **exports** (**X**)
- **Exports** are functions* that the guest makes available to the host
- **Imports** are host-provided functions* that the guest is allowed to access

*: Wasm supports additional import/export types (memories, tables, globals). This presentation focuses solely on function imports and exports

Host/Guest Interaction – Memory



- The host can **freely access and modify** the linear memory of the modules*
- The guest **cannot access host memory** directly – by design, Wasm does not offer instructions to dereference host memory

Glimpse at the Wasm SDK – Module Logic

```rust
#![no_std] // required for wasm32-unknown-unknown
#![feature(alloc_error_handler)]
pub const HEAP_SIZE: usize = 8_000;
define_alloc_heap!(HEAP_SIZE);
define_panic_handlers!();

use module_examples_common::Counter;
use wasm_sdk::{Result, define_alloc_heap, define_panic_handlers, receive, send};
use wasm_sdk_macros::task_run;

#[task_run(inputs(wasm_input), outputs(wasm_data))]
fn run() -> Result<()> {
    let mut buffer = [0; 50];
    loop {
        receive(&mut buffer, Input::WasmInput)?;
        let mut counter = Counter::from_payload(&buffer)?;
        counter.increment();
        let n_write = counter.serialize_into(&mut buffer)?;
        send(&buffer[0..n_write], Output::WasmData)?;
    }
}

```

Glimpse at the Wasm SDK – Module Logic

```rust
#![no_std] // required for wasm32-unknown-unknown
#![feature(alloc_error_handler)]
pub const HEAP_SIZE: usize = 8_000;
define_alloc_heap!(HEAP_SIZE);
define_panic_handlers!();

use module_examples_common::Counter;
use wasm_sdk::{Result, define_alloc_heap, define_panic_handlers, receive, send};
use wasm_sdk_macros::task_run;

#[task_run(inputs(wasm_input), outputs(wasm_data))]
fn run() -> Result<()> {
    let mut buffer = [0; 50];
    loop {
        receive(&mut buffer, Input::WasmInput)?;
        let mut counter = Counter::from_payload(&buffer)?;
        counter.increment();
        let n_write = counter.serialize_into(&mut buffer)?;
        send(&buffer[0..n_write], Output::WasmData)?;
    }
}
```

- no_std
- Explicit allocation

- No "unsafe"
- No Wasm specifics
- Idiomatic error handling
- Focus on domain logic

Glimpse at the Wasm SDK – Host Imports

```rust
#![no_std] // required for wasm32-unknown-unknown
#![feature(alloc_error_handler)]
pub const HEAP_SIZE: usize = 8_000;
define_alloc_heap!(HEAP_SIZE);
define_panic_handlers!();

use module_examples_common::Counter;
use wasm_sdk::{Result, define_alloc_heap, define_panic_handlers, receive, send};
use wasm_sdk_macros::task_run;

#[task_run(inputs(wasm_input), outputs(wasm_data))]
fn run() -> Result<()> {
    let mut buffer = [0; 50];
    loop {
        receive(&mut buffer, Input::WasmInput)?;
        let mut counter = Counter::from_payload(&buffer)?;
        counter.increment();
        let n_write = counter.serialize_into(&mut buffer)?;
        send(&buffer[0..n_write], Output::WasmData)?;
    }
}
```

Glimpse at the Wasm SDK – Host Imports

```rust
#[allow(clippy::cast_sign_loss)]
pub fn receive(buffer: &mut [u8], input_idx: impl Into<InputIdx>) -> ApiResult<usize> {
    let idx: c_int = input_idx.into().into();
    // All other invariants here have to be upheld by the host; Not much we can do on our side
    let status_code =
        unsafe { c_functions::receive_input(buffer.as_mut_ptr(), buffer.len() as c_int, idx) };
    match status_code {
        n_written if n_written >= 0 => Ok(n_written as usize),
        error_code => Err(error_code.into()),
    }
}
```

```rust
#![no_std] // required for
#![feature(alloc_error_hand
pub const HEAP_SIZE: usize
define_alloc_heap!(HEAP_SIZ
define_panic_handlers!();

use module_examples_common:
use wasm_sdk::{Result, de
use wasm_sdk_macros::task_r

#[task_run(inputs(wasm_inpu
fn run() -> Result<()> {
    let mut buffer = [0; 50
    loop {
        receive(&mut buffer, Input::WasmInput)?;
        let mut counter = Counter::from_payload(&buffer)?;
        counter.increment();
        let n_write = counter.serialize_into(&mut buffer)?;
        send(&buffer[0..n_write], Output::WasmData)?;
    }
}
```

MYRMIC
Run anywhere

Glimpse at the Wasm SDK – Host Imports

```rust
#[no_std] // required for
#![feature(alloc_error_hand
pub const HEAP_SIZE: usize
define_alloc_heap!(HEAP_SIZ
define_panic_handlers!();

use module_examples_common:
use wasm_sdk::{Result, defi
```

```rust
#[allow(clippy::cast_sign_loss)]
pub fn receive(buffer: &mut [u8], input_idx: impl Into<InputIdx>) -> ApiResult<usize> {
    let idx: c_int = input_idx.into().into();
    // All other invariants here have to be upheld by the host. Not much we can do on our side
    let status_code =
        unsafe { c_functions::receive_input(buffer.as_mut_ptr(), buffer.len() as c_int, idx) };
    match status_code {
        n_written if n_written >= 0 => Ok(n_written as usize),
```

```rust
#[link(wasm_import_module = "connectors")]
unsafe extern "C" {
    /// Blocking receive: Blocks on the specified input. When an input is available,
    /// the host writes it into the provided buffer (given that it does not exceed
    /// the provided length)
    pub(super) fn receive_input(buffer: *mut u8, length: c_int, input_idx: c_int) -> c_int;
}
```

MYRMIC
Run anywhere

Glimpse at the Wasm SDK – Error Handling

```rust
#![no_std] // required for wasm32-unknown-unknown
#![feature(alloc_error_handler)]
pub const HEAP_SIZE: usize = 8_000;
define_alloc_heap!(HEAP_SIZE);
define_panic_handlers!();

use module_examples_common::Counter;
use wasm_sdk::{Result, define_alloc_heap, define_panic_handlers, receive, send};
use wasm_sdk_macros::task_run;

#[task_run(inputs(wasm_input), outputs(wasm_data))]
fn run() -> Result<()> {
    let mut buffer = [0; 50];
    loop {
        receive(&mut buffer, Input::WasmInput)?;
        let mut counter = Counter::from_payload(&buffer)?;
        counter.increment();
        let n_write = counter.serialize_into(&mut buffer)?;
        send(&buffer[0..n_write], Output::WasmData)?;
    }
}

```

Glimpse at the Wasm SDK – Error Handling

```rust
#![no_std] // required for wasm32-un
#![feature(alloc_error_handler)]
pub const HEAP_SIZE: usize = 8_000;
define_alloc_heap!(HEAP_SIZE);
define_panic_handlers!();

use module_examples_common::counter;
use wasm_sdk::{Result, define_alloc_
use wasm_sdk_macros::task_run;

#[task_run(inputs(wasm_input), outpu
fn run() -> Result<()> {
    let mut buffer = [0; 50];
    loop {
        receive(&mut buffer, Input::
        let mut counter = Counter::f
        counter.increment();
        let n_write = counter.serial
        send(&buffer[0..n_write], Ou
    }
}
```

```rust
#[unsafe(no_mangle)]
pub extern "C" fn run_task() -> i32 {
    let module_result = run();
    match module_result {
        Ok(()) => 0,
        Err(err) => {
            let err_msg = alloc::__export::must_use({
                alloc::fmt::format(alloc::__export::format_args!("Task run error: {err}"))
            });
            ::wasm_sdk::report_error(&err_msg).unwrap();
            wasm_sdk::log(
                &alloc::__export::must_use({
                    alloc::fmt::format(alloc::__export::format_args!("{err_msg}"))
                }),
                wasm_sdk::LogLevel::Error,
            )
            .unwrap();
            -1
        }
    }
}
fn run() -> Result<()> {
    let mut buffer = [0; 50];
    loop {
        receive(&mut buffer, Input::WasmInput)?;
        let mut counter = Counter::from_payload(&buffer)?;
        counter.increment();
        let n_write = counter.serialize_into(&mut buffer)?;
        send(&buffer[0..n_write], Output::WasmData)?;
    }
}
```

MYRMIC
Run anywhere

Developer Ergonomics: What Changes with Wasm?

Constraints

`no_std` environment:

- Modules require an allocator
- Explicit host functions required for system services

ABI boundary:

- Imports/Exports use the C ABI

No rich types at the boundary

- Complex data passed via linear memory

Rust to the Rescue

- Safe wrappers
- Zero-cost abstractions
- Macros / codegen

What we need from a Wasm Runtime

Runtime Requirements on Microcontrollers

In the specific context of our use case, several factors introduce specific requirements:

- Targeting ESP/Nordic microcontrollers
  - Severe limitations in RAM and Flash → **Memory footprint** one of the main concerns
  - Runtime must operate on **bare metal** (without an OS)
  - Runtime must **integrate well with peripheral drivers**
- Flexible deployment and lifecycle control for modules
  - Module **execution must be controllable/preemptable**
- We work with a Rust-based stack
  - **Runtimes written in Rust preferable** due to easier integration and debugging

Runtime Evaluation – Memory Benchmark

Benchmark Code:

Goal:

- Measure memory footprint of different runtimes on a bare metal embedded target

Setup:

- Minimal Wasm module, using a single **synchronous** host function for basic logging
- Module statically loaded into memory (adds 704 Bytes to the footprint which we could avoid)
- All implementations based on a Rust firmware embedding the runtime under investigation

Metric:

- **Overall size of the memory flashed onto the target (a Nordic nrf53 dev board)**

Objective baseline for selecting a runtime suitable for constrained embedded devices.

Runtime Evaluation – Evaluated Runtimes

Runtime	Memory Footprint	Notes
Wasmi		Has plans to provide a serialization mode which will likely bring the footprint to wasmtime levels
Wasmtime		Will likely have an AoT mode in the future
Tinywasm		We are already using serialized modules here
Wamr (interpreter)		
Wamr (AoT)		

Runtime Evaluation – Benchmark Results

Runtime	Memory Footprint	Notes
Wasmi	**592 KiB**	Has plans to provide a serialization mode which will likely bring the footprint to wasmtime levels
Wasmtime	308 KiB	Will likely have an AoT mode in the future
Tinywasm	188 KiB	We are already using serialized modules here
Wamr (interpreter)	80 KiB	
Wamr (AoT)	**68 KiB**	

Runtime Evaluation – Integration w. Peripherals

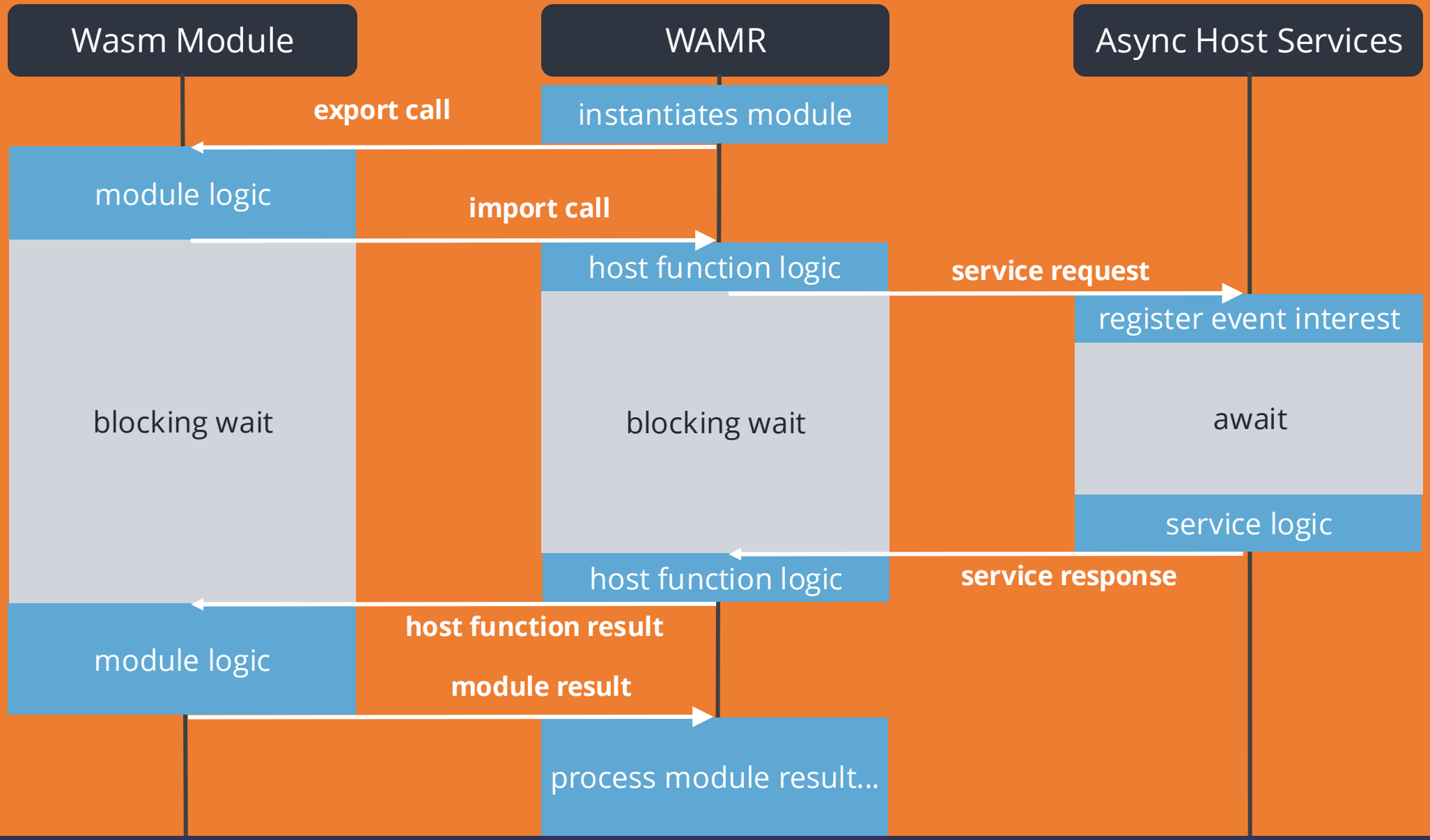
Context:

- Embedded applications typically showcase an intensive interaction with hardware peripherals
- Most operations are therefore IO-bound and must frequently await hardware events
- Implementing non-trivial logic requires the ability to **await multiple independent events concurrently** (timers, sensors, interrupts, communication peripherals, …).

In the Rust embedded ecosystem, this is addressed by the **async machinery** provided by the **embassy framework**.

(Embassy also offers a large amount of other functionality like synchronization- and hardware primitives, all requiring the async API)

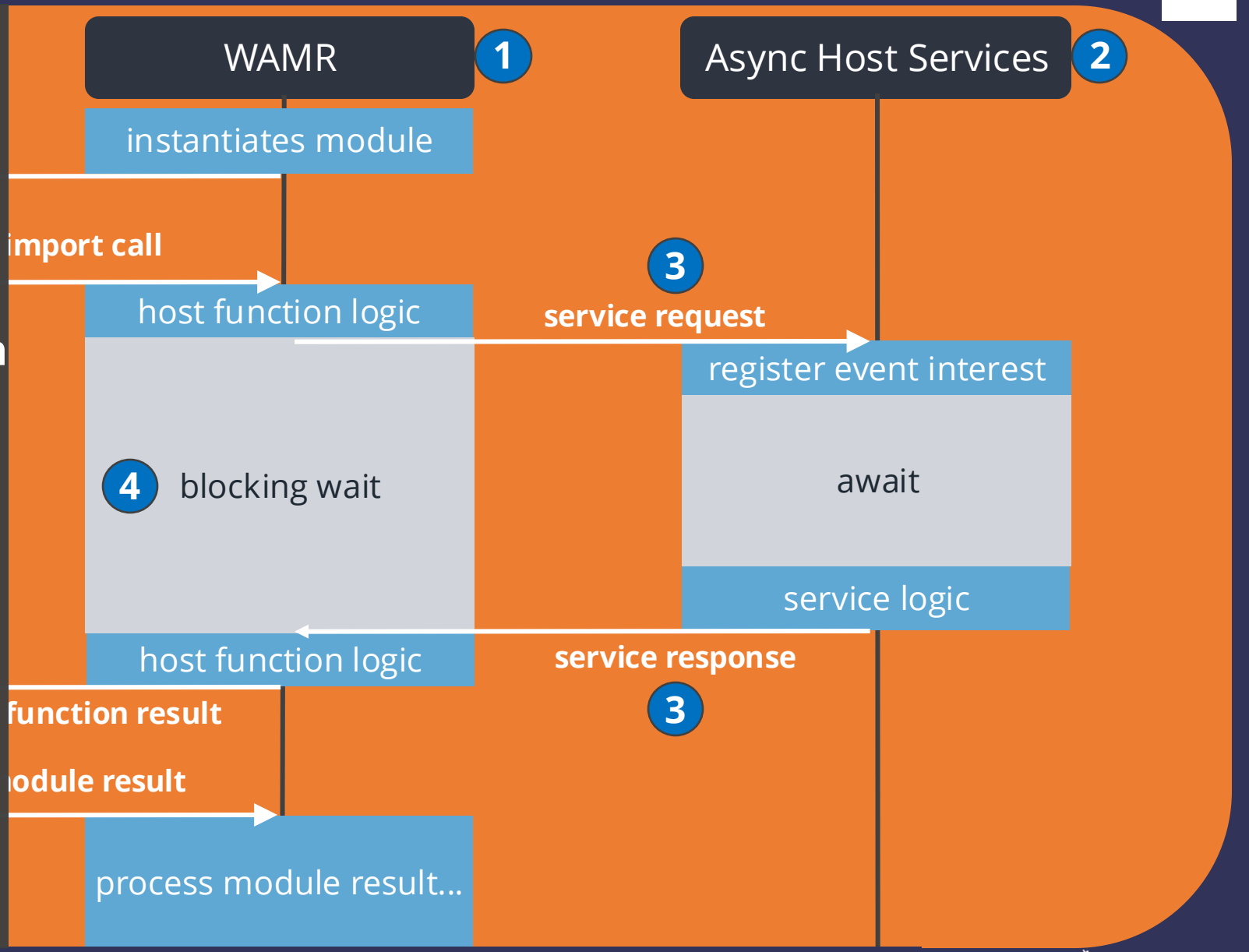
Bringing WebAssembly to constrained devices with Rust



Bringing WebAssembly to constrained devices with Rust

On OS-grade devices, this would be commonly implemented as a SYNC-ASYNC BRIDGE:

1. **WAMR runs in an own thread**
2. **Async Services spawned as an async task**
3. **Communication realized via channels**
4. **When waiting, WAMR blocks its thread to be woken up by the OS when the service response is available**

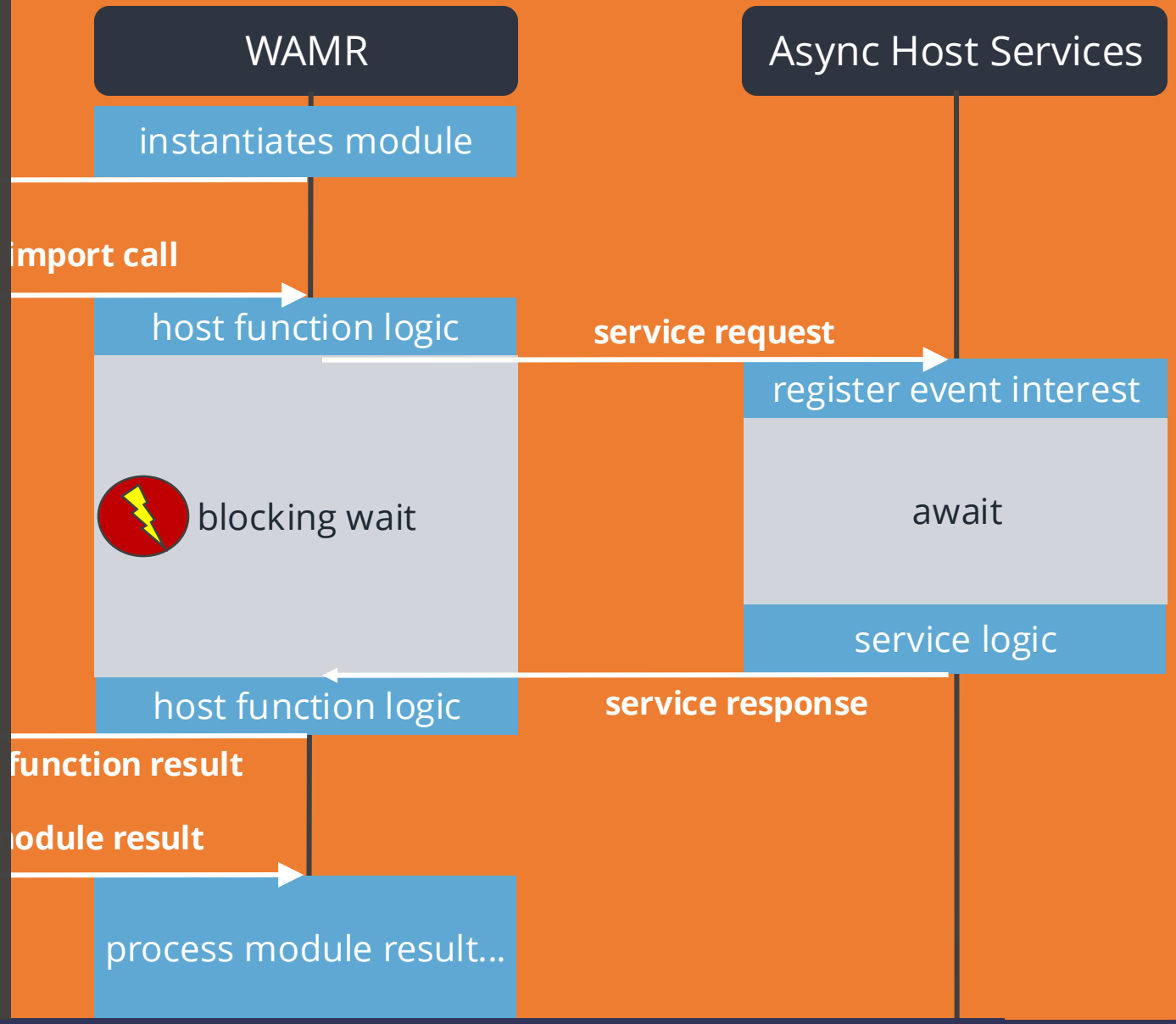


MYRMIC
Run anywhere

On Microcontrollers, this is far less trivial:

We have no threads

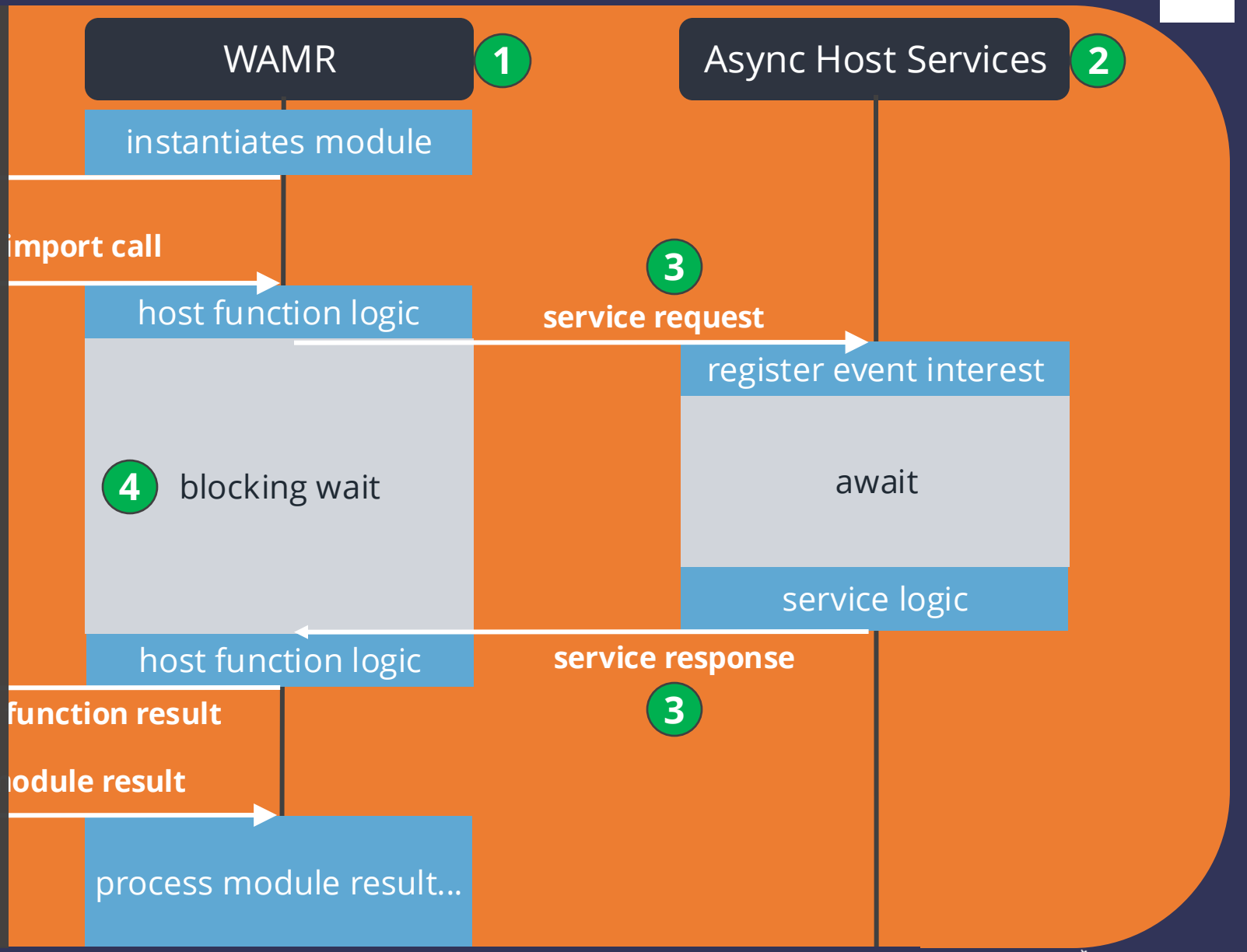
When blocking to wait for the service response, WAMR would block the async host services



Bringing WebAssembly to constrained devices with Rust

Clever workaround on Microcontrollers:

1. WAMR runs in a low-priority task
2. Async Services run as a high-priority async task
3. Communication realized via signals
4. WAMR task blocks using WFI
   - ✅ No busy waiting
   - ✅ Not blocking the async host services
   - ✅ Woken up on response



Where to Go Next

Code & Projects

- Runtime footprint benchmark (nrf53)
  - runs out-of-the box
  - wasmtime, wasmi, tinywasm, WAMR

- Myrmic distributed middleware
  - Open-sourcing later this year
  - Infos at https://myrmic.org/

Credits

WARM async integration -- Alessandro Gasbarroni

- GitHub: `lakier15`
- Author of the `c-compat` crate

MYRMIC
Run anywhere