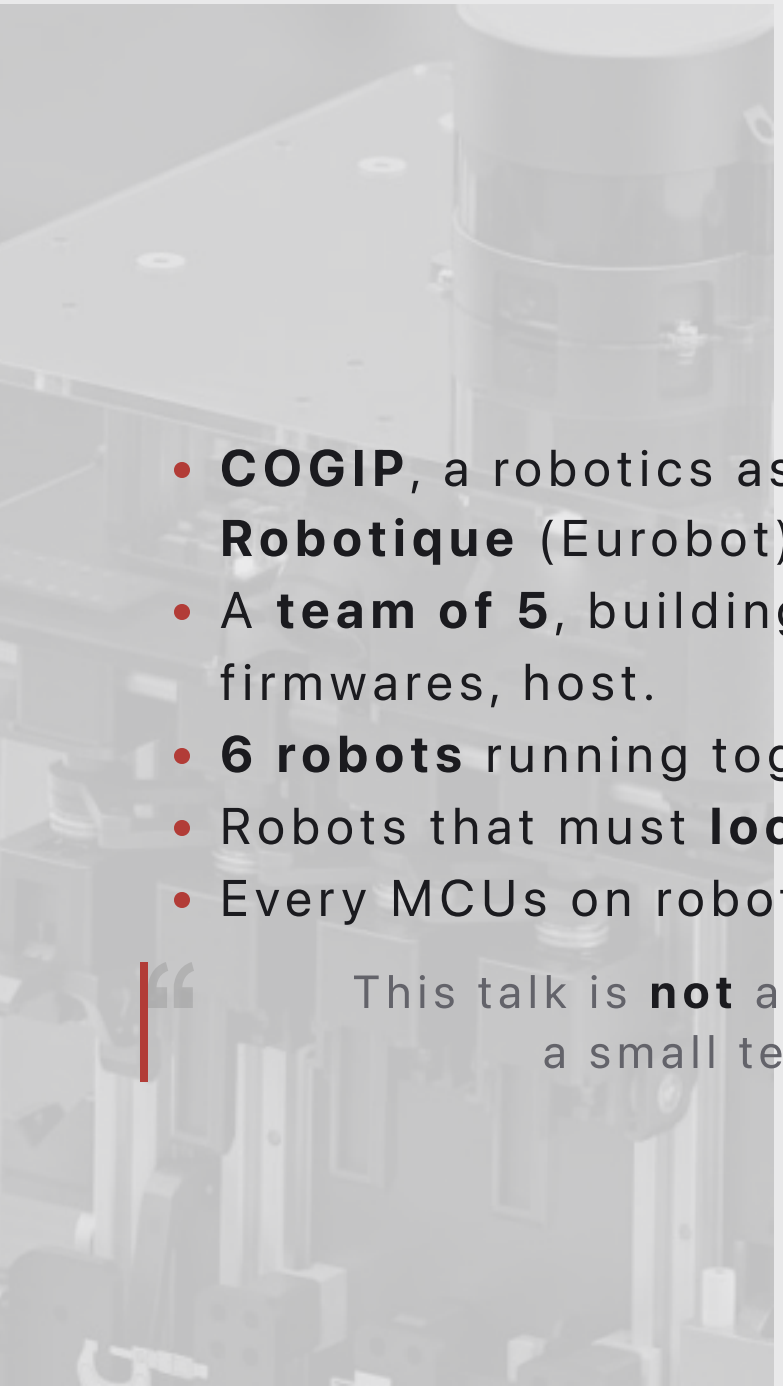



# A real robot firmware on RIOT-OS

From a custom STM32 board to over-the-network updates:
everything we ship on RIOT.

**RIOT Summit 2026**

Gilles DOFFE & Mathis LECRIVAIN

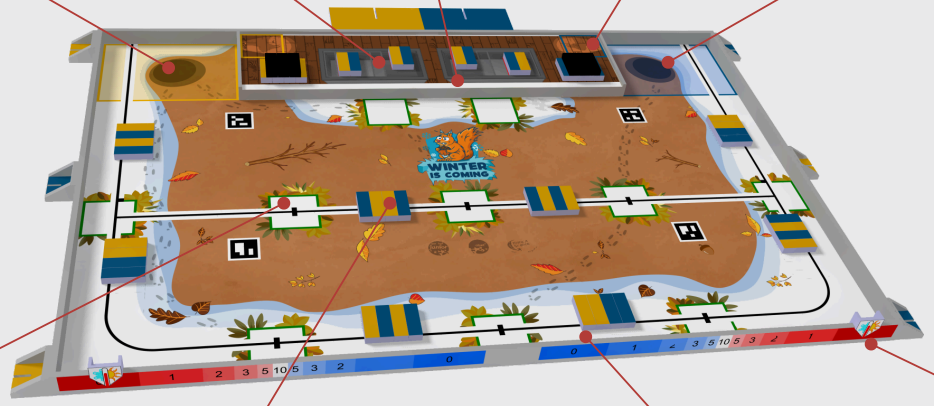


# Who we are

- **COGIP**, a robotics association competing in the **Coupe de France de Robotique** (Eurobot).
- A **team of 5**, building the whole robots: mechanics, electronics, firmwares, host.
- **6 robots** running together on the table: 1 main robot, 4 SIMAs and 1 "Ninja"
- Robots that must **localize, plan and move** autonomously on a table, in 100s.
- Every MCUs on robots runs **RIOT-OS** and is developed in **C++20**.

> This talk is **not** a deep dive into one feature, but a **map** of what RIOT lets a small team build end to end, focusing on the main robot.

# The playing area

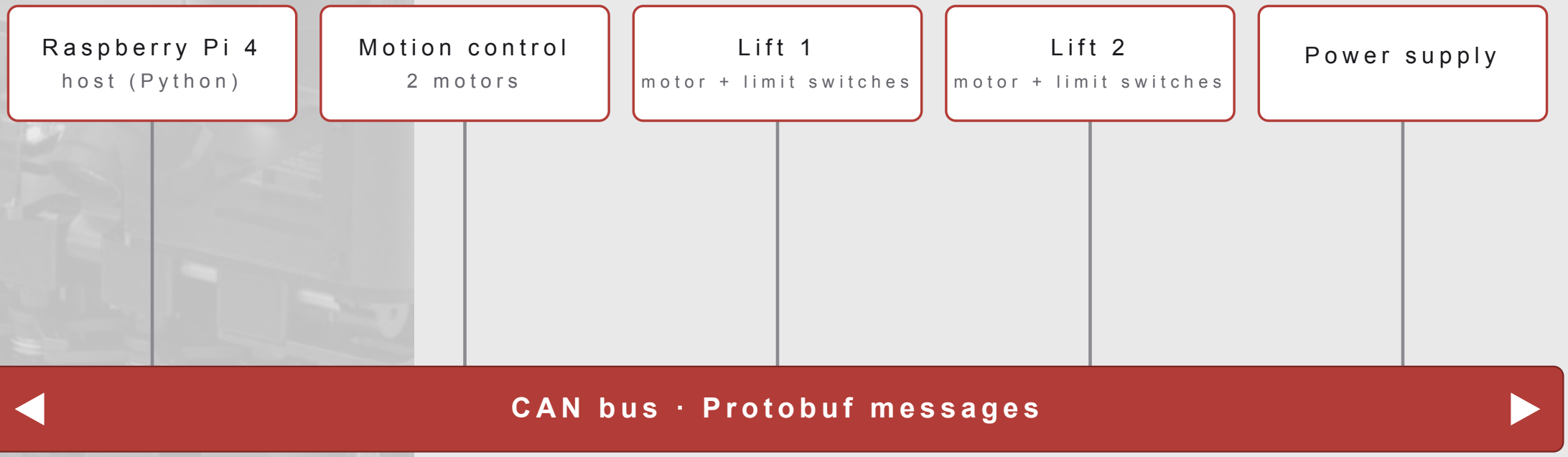



Set cursor, grab, sort and put down crates in given areas while avoiding the opponent, all in **100 s** with **no operator**.

# The software stack

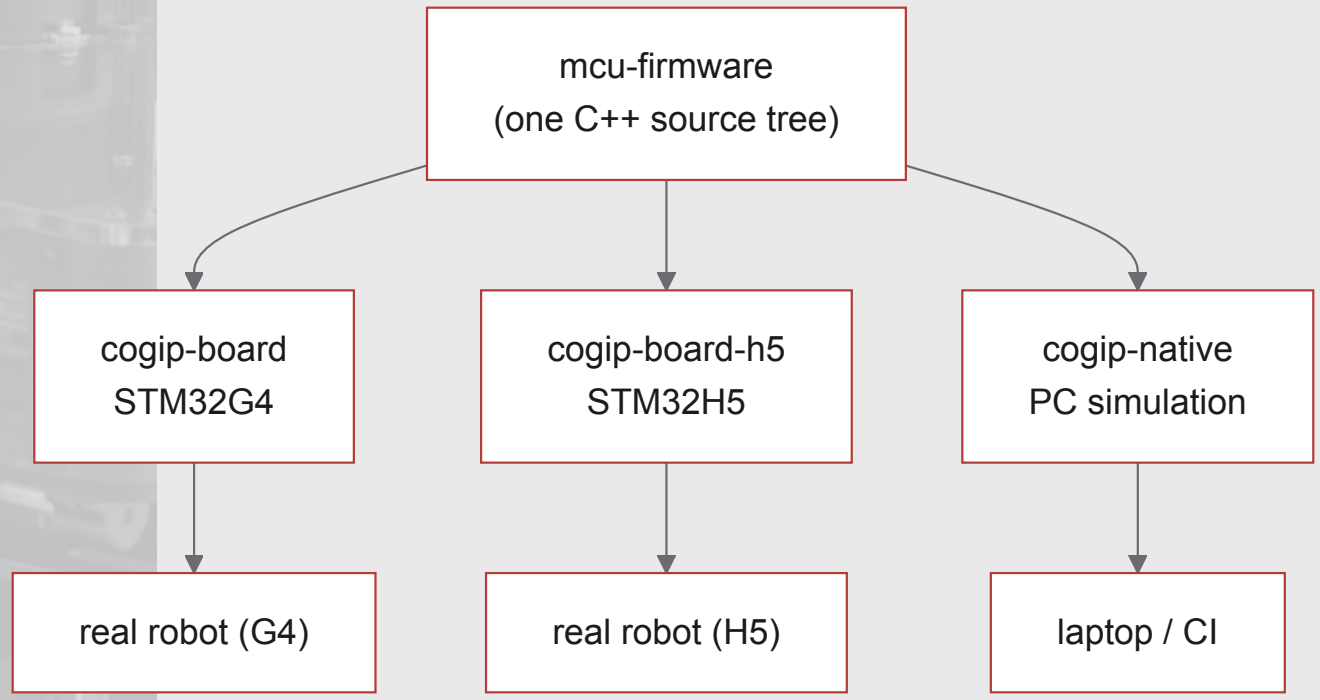
- **cogip-tools**: Python/C++ host on the Raspberry Pi. Planner, copilot, avoidance, dashboard, tools, Protobuf python bindings.
  - `github.com/cogip/cogip-tools`
- **mcu-firmware**: C++ firmware for the STM32 boards. Motion, actuators, sensors, EMS, logs.
  - `github.com/cogip/mcu-firmware`
- **RIOT (our fork)**: the RTOS the firmware runs on. Branch `cogip_master`, board and CPU additions, packages versions updates.
  - `github.com/cogip/RIOT`

# The big picture



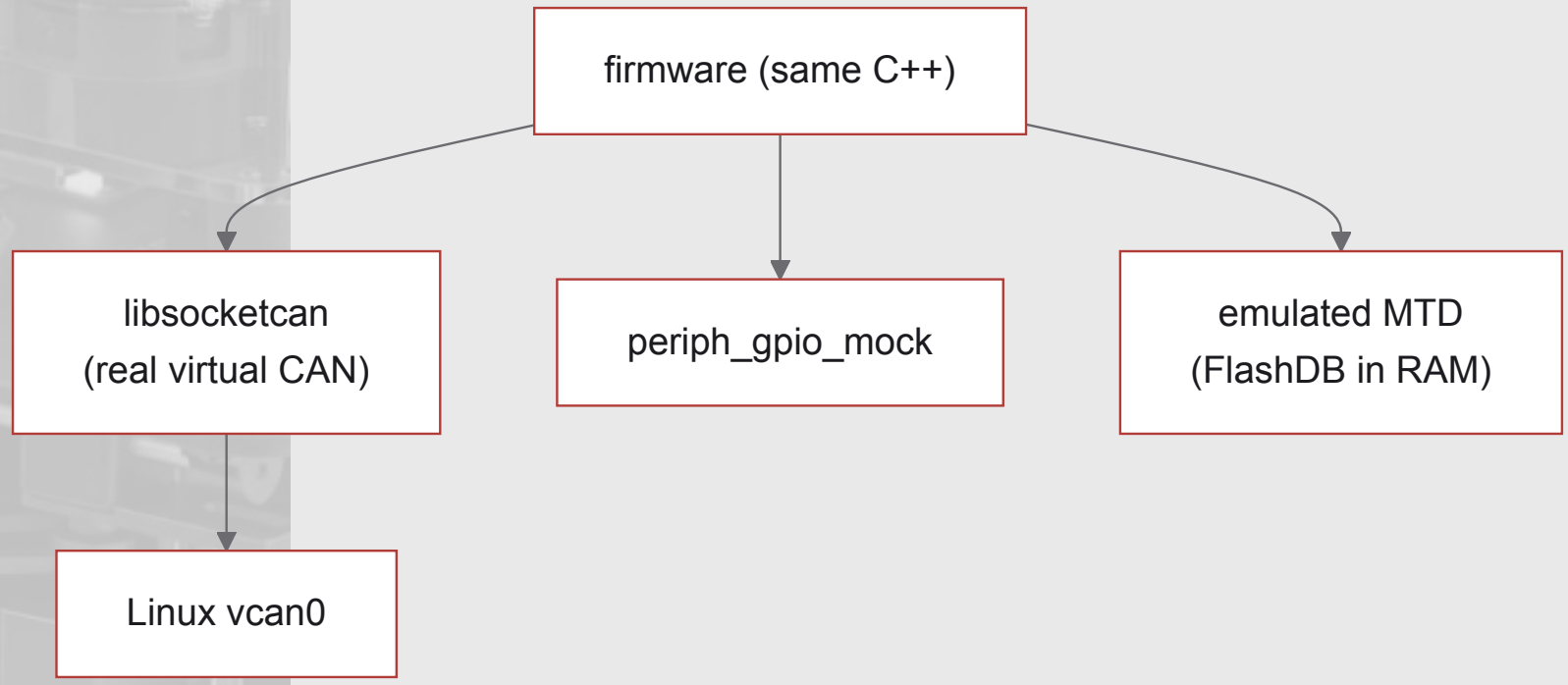

**4 STM32 boards + 1 Pi 4**, all CAN nodes speaking **Protobuf**. Same "motors" board runs three roles (motion + two lifts) on one HW, configured at build time.

# Three targets, one codebase



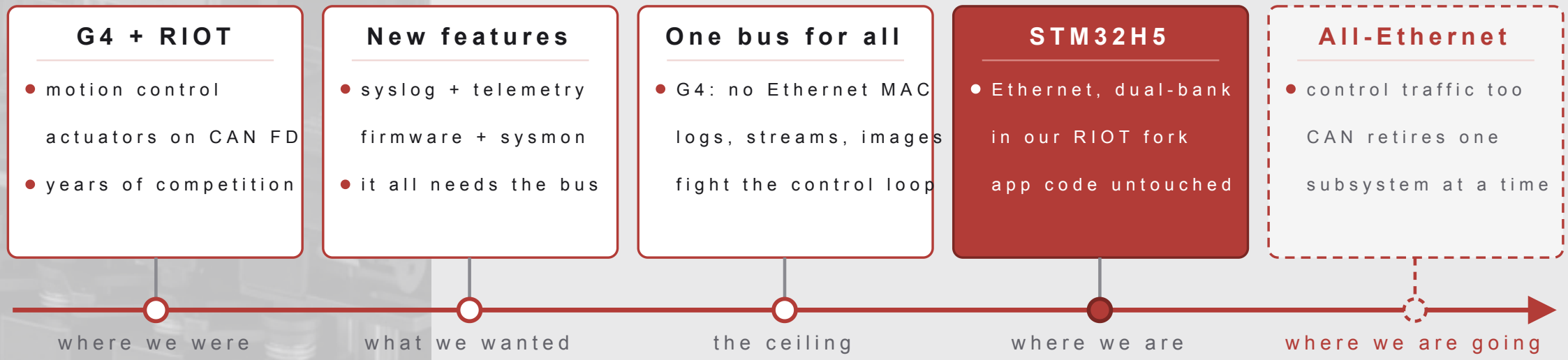

Same controllers, same Protobuf, three back-ends. `make BOARD=...` and go. On `cogip-native`, **docker compose** spins up the full stack (firmware + host + virtual CAN), no hardware.

# Native simulation




`cogip-native` compiles the firmware **as a PC binary**. Real CAN frames on a virtual bus, mocked GPIO, emulated flash. Dev and **CI** with **zero hardware**.

# Why we moved to the H5




**Today** CAN FD carries the control loop and Ethernet takes the new features: logs, telemetry, firmware images. We move **one subsystem at a time**.

# Adding the STM32H5

Brand-new **CPU family** bring-up in our RIOT fork.

### Working now

- CAN (FDCAN)
- Ethernet
- PWM · QDEC
- UART · GPIO · Timer
- riotboot (dual-bank flash)

### What we wrote

- a new **CPU family** in our RIOT fork
- one **thin board** layer, just three files
- RAM-function flash erase (read-while-write)

> Two boards, two CPU families, **one application tree**, and nothing in the port is COGIP-specific, which is why it is worth **upstreaming**.

*Full disclosure: the code is written by Claude. We just do the thinking (a bit).*



# Protobuf on the MCU

- **EmbeddedProto**: Protobuf on a microcontroller.
- Our **own RIOT-style package**, not upstream: `USEPKG += embedded-proto`.
- RIOT's **pkg system is extensible**: external package builds like a built-in, no fork.
- `protoc` runs **at build** and emits **C++ with no heap** (like ETL).
- **Same** `.proto` also generates the **Python** host bindings.

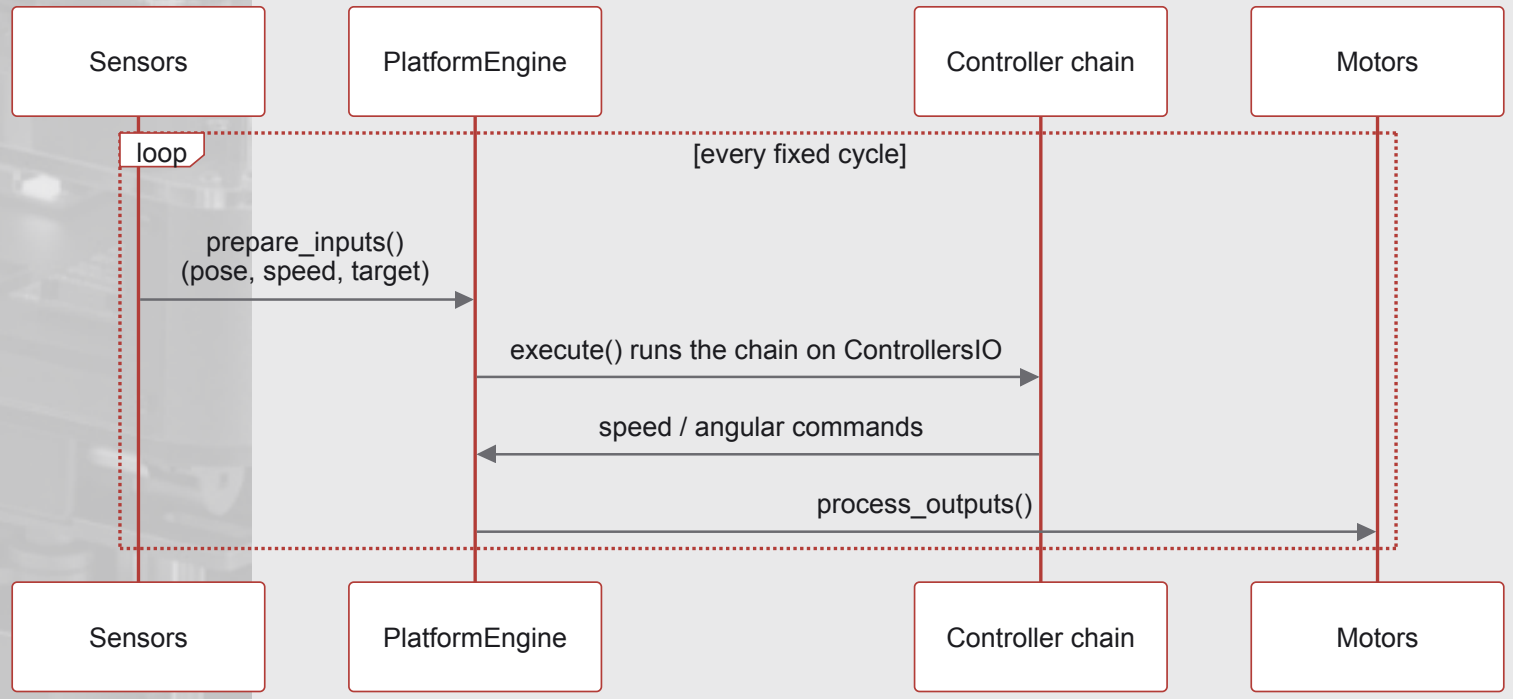
# ETL, the STL without a heap

- Embedded Template Library: containers with **no dynamic allocation**.
- `etl::variant`, `etl::unordered_map`, `etl::string<N>` (bounded, deterministic).
- Example: `ControllersIO`, a typed key-value store passed down the control chain.

```
// value is etl::variant<float, double, int, bool, etl::string<32>>
io.set("linear_pose_error", err);
auto s = io.get_as<float>("linear_current_speed"); // etl::optional<float>
```

We never call `malloc`. Memory use is fixed at build time.

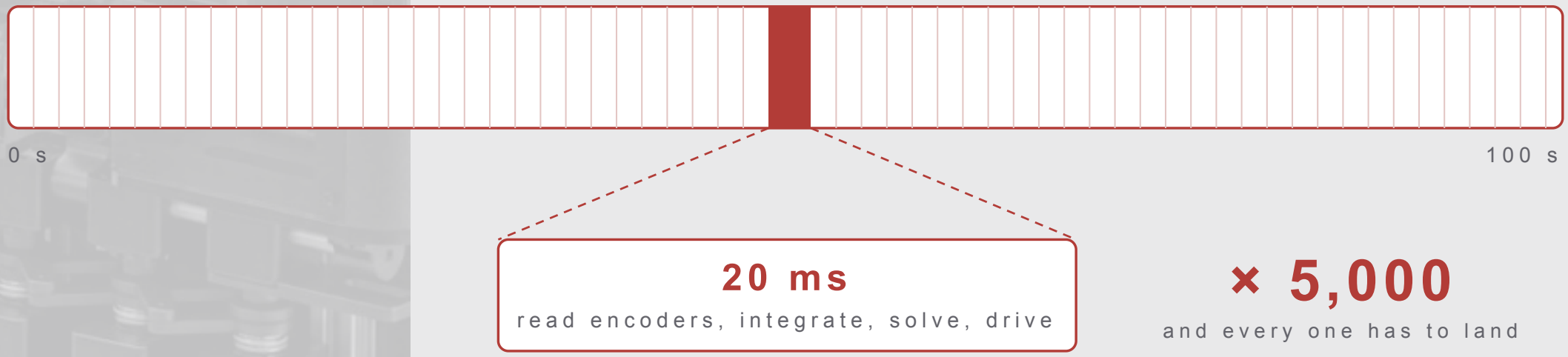
# Motion control, the cycle




The loop runs at a fixed period. As long as it runs, the firmware is **healthy**.
The watchdog checks exactly this (later).

# 100 seconds, 5,000 cycles

one match: fully autonomous, no operator, no retry




**× 5,000**

and every one has to land

**No operator, no retry, no second run.** So the maths runs on the MCU, not over a link. The deadline is hard.

# All of it, on one MCU

**STM32G474RE · 170 MHz Cortex-M4**

- path following (waypoints)
- trapezoidal velocity profiles
- go-straight filter rotate·move·rotate
- cascaded PID pose + speed
- braking-distance limiting
- anti-blocking detection

**50 Hz**

20 ms · drift-corrected ztimer

Flash: 512 KB — **210,672 B · 40.2 %**



RAM: 96 KB SRAM — **70,124 B · 71.3 %**



arm-none-eabi-size, on the binary we compete with

**One engine, several controller chains** (QuadPID, profile tracker, brake), selected at runtime. **The whole firmware fits in 206 KB**, no dynamic allocation.

# Room to grow: the same, on the H5

**STM32H563RI · 250 MHz Cortex-M33**

- path following (waypoints)
- trapezoidal velocity profiles
- go-straight filter rotate·move·rotate
- cascaded PID pose + speed
- braking-distance limiting
- anti-blocking detection

## 50 Hz

20 ms · drift-corrected ztimer

Flash: 2 MB — **239,464 B · 11.4 %**



RAM: 640 KB SRAM — **98,180 B · 15.0 %**



same firmware, arm-none-eabi-size on the H5 build

**Not one line of the control chain changed.** On the **H5**: **12% flash, 15% RAM**, with room for Ethernet, logging, and bigger trajectories.

# Parameters & telemetry

### Tune live

- `parameter_handler`
- change PID gains over CAN/Protobuf, no reflash
- QUADPID and tracker gains tuned **independently**
- persisted in a **FlashDB** key-value store, survives reboots

### Observe

- `telemetry` streams robot state
- pose, speed, controller internals
- live, on a dev PC on the network

> Development loop: run, watch telemetry, retune, repeat, all without a rebuild.

# Observability without a probe

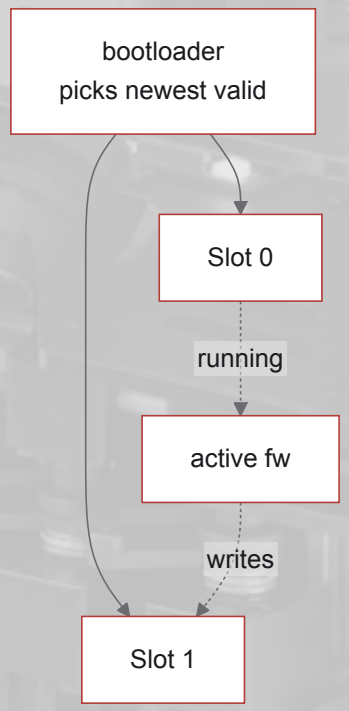
- `telemetry`: live robot state (pose, speed, controller internals) over CAN as Protobuf (over Ethernet soon ;)).
- `sysmon` (thread list, stack usage, heap), all as Protobuf messages.
- **syslog forwarding**, console tee'd to **RFC 5424** over UDP → journald.
- Boot logs captured too (ARP-gated flush so the first lines aren't lost).

```
<14>1 2026-08-25T10:12:03Z robot6 fw - - - boot confirmed healthy
```

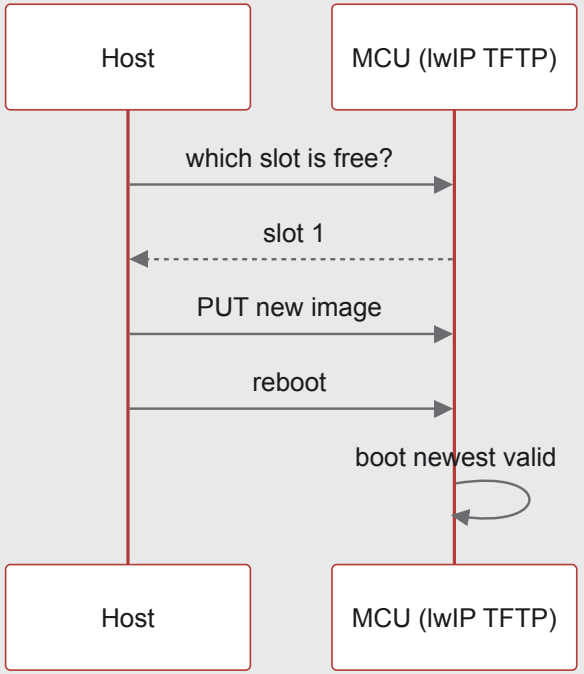
Field debugging over the network, no serial cable, no J-Link.

# Safe field update, end to end

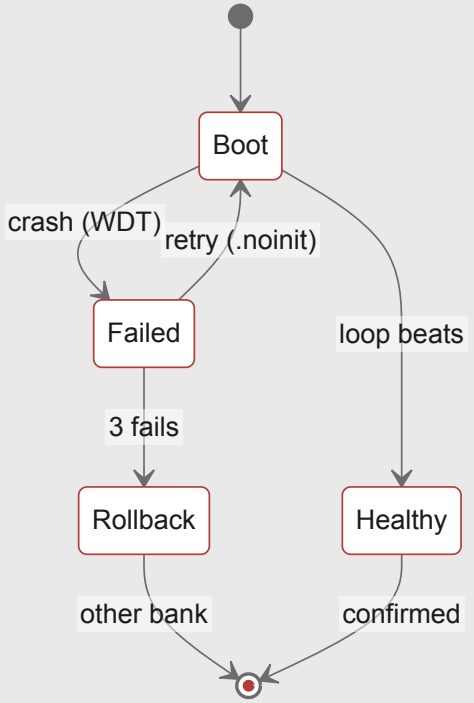
## 1. Dual-bank slots




## 2. Flash over the network



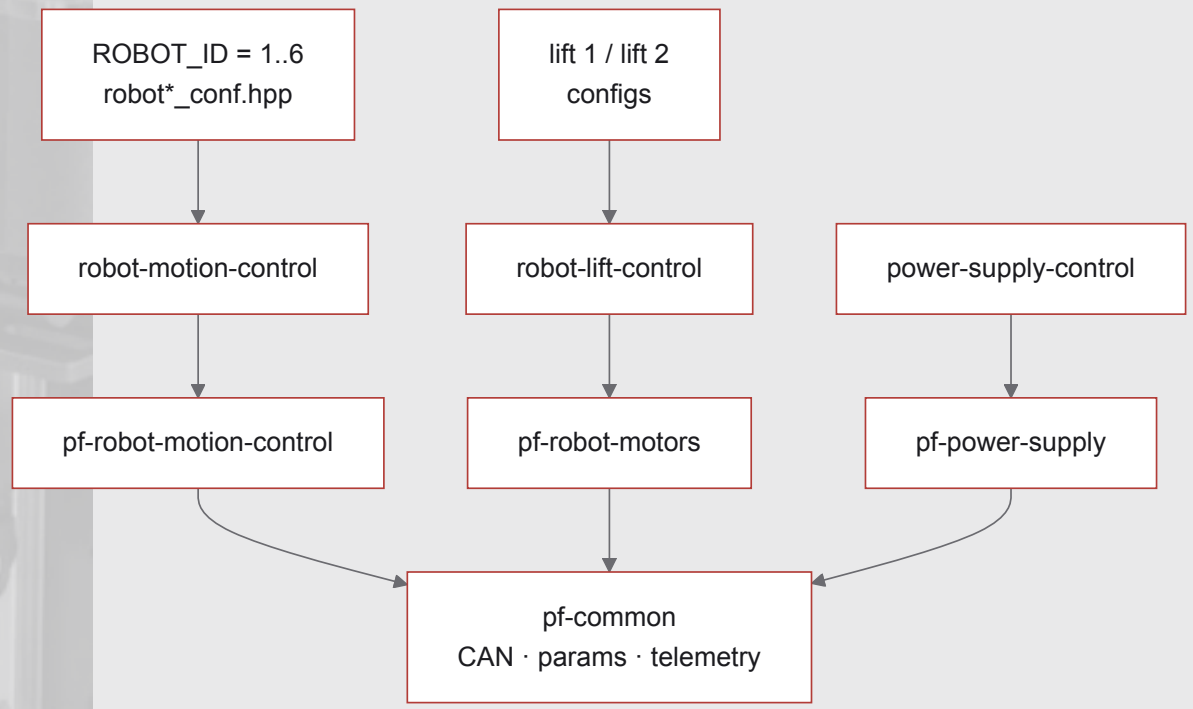

## 3. Safe rollback



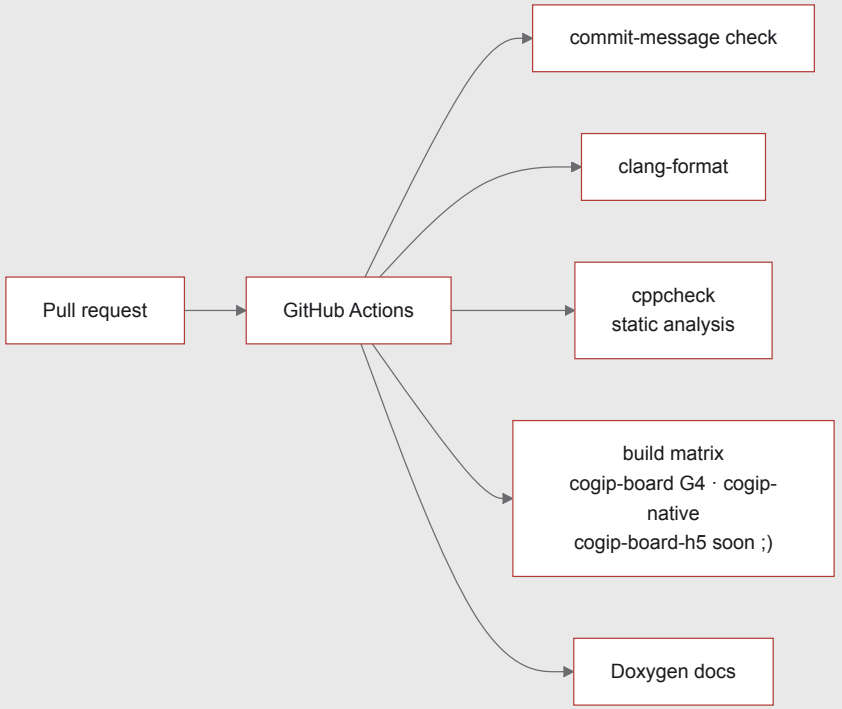

`make flash-net`: push a build over the network, and a crashing image can never brick the robot (rolled back after **3** fails, even across a cold boot).

# Actuators & platforms

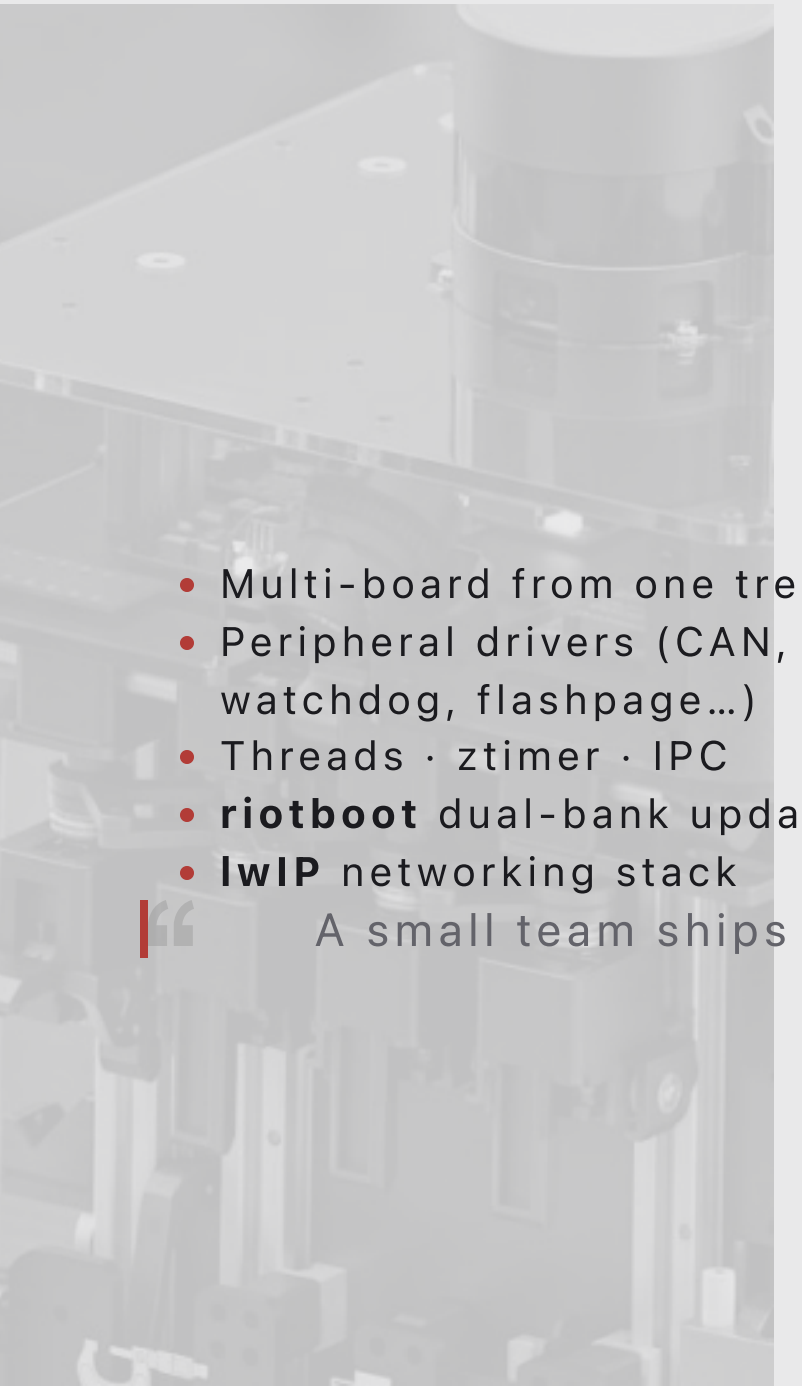
- `lib/actuator` (motors, lifts, limit-switch sensors) behind one interface.
- One app per platform `pf-*`, all sharing **`pf-common`** (CAN, params, telemetry).
- One base, several robots: `robot1` / `robot2` / `robot6` build-time configs.



# Quality & CI




Every PR: commit-message lint, **clang-format**, **cppcheck** static analysis, a **build matrix** (G4 + native, **H5 soon** ;)), and Doxygen.



# What RIOT gives us

- Multi-board from one tree
- Peripheral drivers (CAN, ETH, PWM, QDEC, watchdog, flashpage…)
- Threads · ztimer · IPC
- **riotboot** dual-bank update
- **lwIP** networking stack
- **pkg** system: ETL, EmbeddedProto, FlashDB
- **native** board = test on a laptop
- C++ support + libstdc++ (we bump it to C++20)

> A small team ships a full robot stack because **RIOT already carries the OS**.

# Upstream-able contributions

The whole **STM32H5 support** we integrated (CPU family + peripherals), plus:

- **RAM-function flash erase** for STM32H5 (dual-bank read-while-write).
- `lwip_app_tftp` module, TFTP over lwIP.
- `stdio_syslog`, console tee to RFC 5424 UDP.
- **confirmed-boot / rollback** pattern ( `.noinit` + watchdog + slot validation).

Code generated with **Claude**: six months on a single driver is not an option when you compete. PRs go up in batches only after full **review and tests**, so we keep control of the **what** and the **how**.

# Thank you

**Cogip**, a robot stack, top to bottom, on RIOT-OS.

`github.com/cogip/mcu-firmware` · `github.com/cogip/RIOT`

**Questions?**

WeBotics

RTONE

01 OMIRION

Transformer vos idées en réalité

MaenSkill

Embedded Systems & Training