ExoStack v1.0
User Manual

Supplementary material: installation, configuration, operation and extension of the ESA firmware, the ExoCore ROS 2 package and the PASE graphical interface

Accompanies the article *ExoStack: A Modular and Reconfigurable Software Framework for Wearable Robotics Using ROS 2*, IEEE Robotics and Automation Practice.
Source code: `https://github.com/debadrata/ExoStack` · Documentation website: `https://debadrata.github.io/ExoStack_documentation/`
© 2025–2026 the ExoStack authors, CSIR-CMERI. This document is licensed under CC BY-NC 4.0; the ExoStack code under the PolyForm Noncommercial License 1.0.0.

Contents of the supplementary material

Item	Folder / file	Purpose
Code	`ESA/`	ESP32 firmware library (micro-ROS) and the vendor libraries it depends on
Code	`exo_stack/`	ExoCore, the ROS 2 Python package: synchronizer, gait-event, planner, profile, cost and HIL nodes
Code	`exo_stack_resources/`	configuration files, algorithm plug-ins, lookup tables, HIL results
Code	`PASE_GUI/`	desktop application to configure, launch and visualize the stack
Manual §1	this document	repository layout
Manual §2.1	this document	installation of ESA, the micro-ROS agent, ExoCore and PASE
Manual §2.2	this document	configuration: ESA sketch, YAML files, lookup tables, by hand or in PASE
Manual §2.3	this document	running the stack: from the PASE Launch panel or from the terminal
Manual §2.4	this document	the PASE Visualization panel
Manual §2.5	this document	ESA: adding an I²C sensor driver
Manual §2.6	this document	ESA: adding an actuator driver
Manual §2.7	this document	ExoCore: adding gait-event, builder, lookup, optimizer, cost and planner plug-ins
Manual §3	this document	benchmarking: set-up and results on three host computers
Manual §4	this document	configurations of the two exosuits on which ExoStack was validated
Manual App. A	this document	troubleshooting, and the ROS 2 and micro-ROS interfaces (Table S3)

Conventions used in this manual

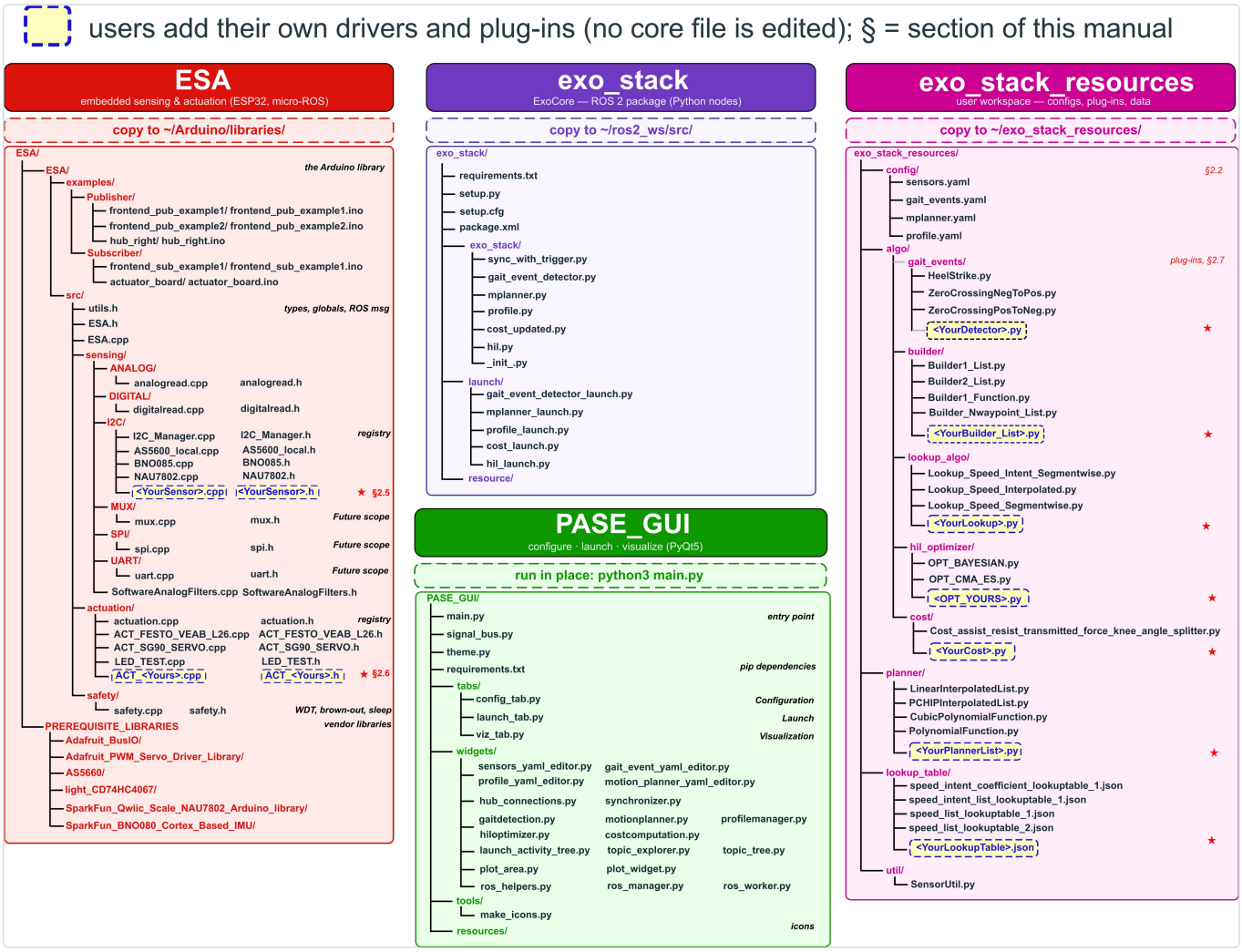
Notation	Meaning
`$ command`	type in a terminal (do not type the `$`)
`~/exo_stack_resources`	a path; `~` is your home directory
`MYSENS` 1	in a code template: **change this**; the number refers to the explanation below the template
◀ 1	arrow pointing at a line you edit

WORKED EXAMPLE USED THROUGHOUT

One configuration runs through every section so that the files stay consistent with each other: a **right-side sensor hub** publishing 21 channels on `/hub_right` (two 9-channel IMUs, two load cells, one foot-pressure sensor), a mirror **left hub** on `/hub_left`, and one **actuation board** driving a pneumatic valve per side on `/right/hip_ppr_cmd` and `/left/hip_ppr_cmd`. These are the configurations shipped in `exo_stack_resources/config`.

1 Repository layout

ExoStack has four parts. Each is installed in a different place; the column header of the figure gives the destination, and the highlighted boxes mark where you add your own drivers and algorithms.



Repository layout of ExoStack v1.0 (Fig. 6 of the article). Highlighted entries are extension points; the § mark gives the section of this manual that explains them.

Part	Runs on	Language	Installed to	Role
ESA	ESP32 boards on the body	C++ (Arduino)	`~/Arduino/libraries/ESA`	reads sensors and drives actuators; publishes/subscribes through micro-ROS
ExoCore (`exo_stack`)	host PC / Jetson	Python, ROS 2	`~/ros2_ws/src/ExoStack/exo_stack` (cloned)	synchronizes hubs, detects gait events, builds and plays assistance profiles, computes cost, runs HIL
exo_stack_resources	host	YAML, JSON, Python	`~/exo_stack_resources`	everything you edit: configuration, plug-ins, lookup tables; read by ExoCore at start-up
PASE	host	Python, PyQt5	anywhere (run in place)	edits the configuration, starts/stops every stage, plots any topic

FIXED LOCATION

ExoCore and PASE look for `exo_stack_resources` **directly in your home directory**. It must be `~/exo_stack_resources` , not a sub-folder and not inside the ROS workspace.

2.1 Installation

Four installations, in this order: the ESA firmware environment (Arduino), the micro-ROS agent (host), ExoCore (host) and PASE (host). Allow about 30 minutes.

Requirements

Item	Version	Notes
Host OS	Ubuntu 22.04	on a PC or NVIDIA Jetson
ROS 2	Humble (desktop)	`ros-humble-desktop` provides `message_filters` and `sensor_msgs`
Python	3.10	system Python of Ubuntu 22.04
Arduino IDE	2.x	
ESP32 board package	2.0.17 (tested)	Espressif *esp32* core; must match the micro-ROS build below
micro_ros_arduino	2.0.7-humble (tested)	pre-built micro-ROS client for Arduino
Microcontroller	ESP32 (ESP-WROOM-32 tested)	two I²C buses, ADC, DAC

Get the code

One `git clone` fetches all four parts; each is then installed from its own folder. Cloning into the ROS 2 workspace lets `colcon` find ExoCore in place; the folders without a ROS package carry a `COLCON_IGNORE` file.

terminal

```
$ mkdir -p ~/ros2_ws/src && cd ~/ros2_ws/src
$ git clone https://github.com/debadrata/ExoStack.git
$ cd ExoStack                                  # paths below are relative to this folder
```

A. ESA firmware environment

1. Install the **Arduino IDE**. In *Boards Manager*, install **esp32 by Espressif Systems** at the version listed above.
2. Download the Humble release of **micro_ros_arduino** (GitHub: `micro-ROS/micro_ros_arduino`) as a ZIP and add it with *Sketch → Include Library → Add .ZIP Library*.
3. Copy every folder inside `ESA/PREREQUISITE_LIBRARIES_COPY_TO_ARDUINO_LIBRARIES/` into `~/Arduino/libraries/`.
4. Copy the library folder `ESA/ESA/` into `~/Arduino/libraries/` so that you obtain `~/Arduino/libraries/ESA/src/ESA.h`.
5. Restart the IDE. Open *File → Examples → ESA → Publisher → hub_right*, select your ESP32 board and press **Verify**. A clean compile confirms the installation.

Library (copied in step 3)	Version	Used by
AS5600	0.6.7	magnetic joint encoder driver
Adafruit_BusIO	1.17.4	dependency of the PWM driver
Adafruit_PWM_Servo_Driver_Library	3.0.3	PCA9685 servo driver
SparkFun_BNO080_Cortex_Based_IMU	1.1.12	BNO085 IMU driver
SparkFun_Qwiic_Scale_NAU7802_Arduino_Library	1.0.6	NAU7802 load-cell amplifier driver
light_CD74HC4067	1.0.1	analog multiplexer (reserved for future use)

ONE COPY OF ESA ONLY

Keep exactly one copy of the ESA library in `~/Arduino/libraries`. Remove any older sensor-suite library from earlier releases; two libraries defining the same classes cause duplicate-symbol errors.

B. micro-ROS agent (host)

The agent bridges the ESP32 boards to the ROS 2 graph. PASE starts it for you, but it must be built once.

terminal

```
$ source /opt/ros/humble/setup.bash
$ mkdir -p ~/microros_ws/src && cd ~/microros_ws
$ git clone -b humble https://github.com/micro-ROS/micro_ros_setup.git src/micro_ros_setup
$ sudo apt update && rosdep update && rosdep install --from-paths src --ignore-src -y
$ colcon build && source install/local_setup.bash
$ ros2 run micro_ros_setup create_agent_ws.sh
$ ros2 run micro_ros_setup build_agent.sh
```

C. ExoCore

terminal

```
$ cd ~/ros2_ws/src/ExoStack
$ pip3 install -r exo_stack/requirements.txt      # numpy, scipy, pyyaml, cmaes, bayesian-optimization==1.4.3
$ cp -r exo_stack_resources ~/                    # must end up as ~/exo_stack_resources
$ cd ~/ros2_ws && colcon build --packages-select exo_stack
```

WHY THE VERSION PIN

`OPT_BAYESIAN` uses the acquisition-function interface of *bayesian-optimization* 1.4. Version 2.0 removed it.

D. PASE GUI

terminal

```
$ cd ~/ros2_ws/src/ExoStack/PASE_GUI
$ pip3 install -r requirements.txt                # PyQt5, pyqtgraph, ruamel.yaml, ...
$ python3 main.py                                 # run PASE from this folder
```

E. Shell environment

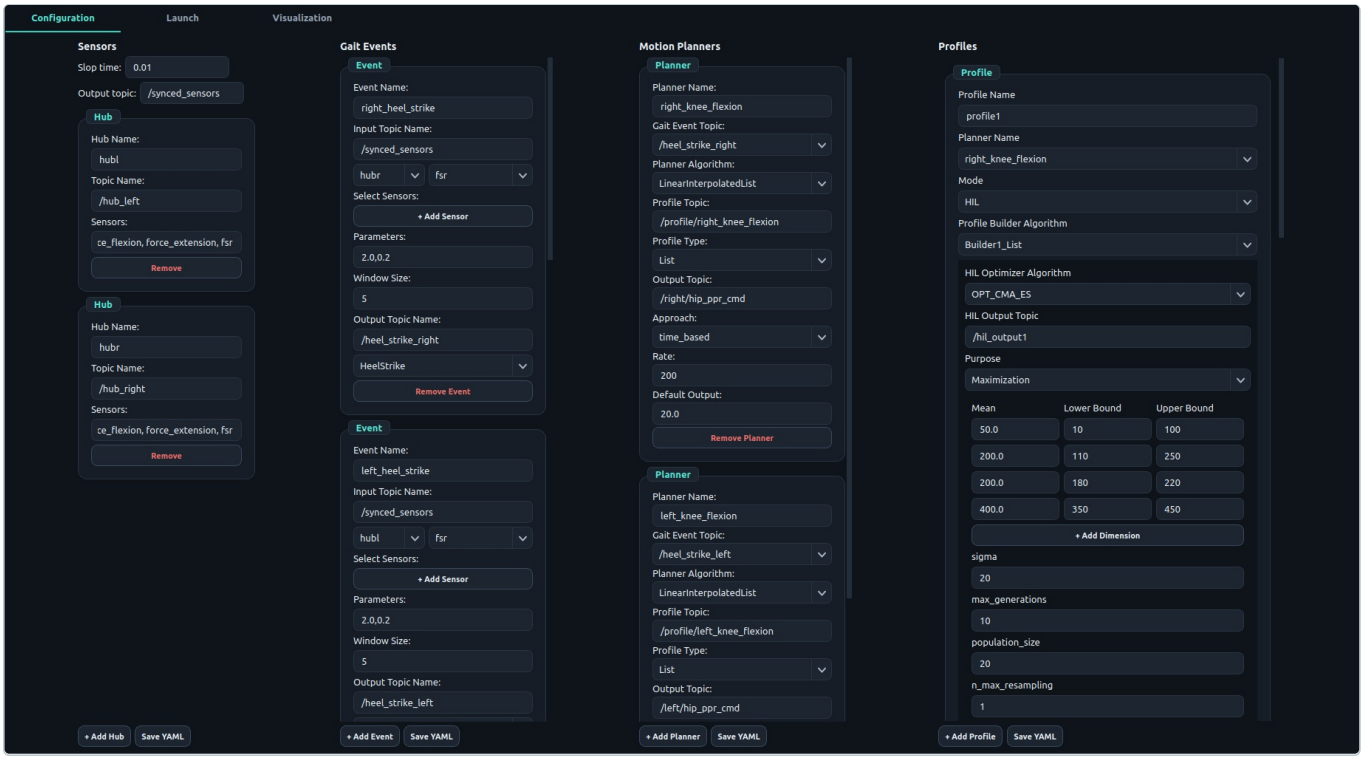
Every terminal that runs ExoCore or PASE needs all three environments. Append to `~/.bashrc` once:

`~/.bashrc`

```
source /opt/ros/humble/setup.bash
source ~/microros_ws/install/local_setup.bash
source ~/ros2_ws/install/setup.bash
```

F. Verify

Check	Command	Expected
ExoCore executables	`ros2 pkg executables exo_stack`	includes `sync`, `gait_event_detector`, `mplanner`, `actuation_profile`, `cost`, `hil`
Launch files	`ros2 launch exo_stack` + Tab	five `*_launch.py` files
Resources in place	`ls ~/exo_stack_resources/config`	`gait_events.yaml mplanner.yaml profile.yaml sensors.yaml`
Agent	`ros2 run micro_ros_agent micro_ros_agent --help`	usage text
PASE	`cd PASE_GUI && python3 main.py`	window with Configuration, Launch, Visualization tabs



Screenshot S2.1-1. PASE main window on the Configuration tab: the Sensors, Gait Events, Motion Planners and Profiles panels, filled from the YAML files.

2.2 Configuration

A complete setup is six small files that refer to each other by name. Configure them left to right; each one uses names defined by the one before it.

```
ESA sketch ──► sensors.yaml ──► gait_events.yaml ──► mplanner.yaml ──► profile.yaml ◄── lookup_table/*.json
 (per board)     (hubs, channels)    (event detectors)    (planners)        (profile source, cost, HIL)
```

This value …	… must equal this value
sketch `setupPublisher(`**`topic`**`, …)`	`sensors.yaml → hubs.<hub>.topic`
order and count of channels the board publishes	order and count of names in `hubs.<hub>.sensors`
`gait_events.yaml → <event>.output_topic`	`mplanner.yaml → <planner>.gait_event_topic`
`mplanner.yaml` planner key	`profile.yaml → <profile>.planner_name`
`mplanner.yaml → <planner>.output_topic`	sketch `setupActuation(`**`topic`**`, …)` on the actuation board
algorithm names in any YAML	a file of that name (and a class of the same name) in its plug-in folder

A. ESA sketch: sensing hub

A board is *declared*, not programmed: list the devices, then call the set-up functions in the order shown. Highlighted values are the ones you change.

hub_right.ino

```
#include "ESA.h"
SoftExoWalk exo;

DeviceConfig deviceList[] = {
//  type       addr/pin   name       extra_info1     misc1         misc2
  { I2C[1], 0x4A[2], "IMU_T"[3], MAIN[4],   9[5] },               // thigh IMU, 9 channels
  { I2C,       0x4A,       "IMU_S",       AUX1,         9 },                      // shank IMU (same address, 2nd bus)
  { I2C,       0x2A,       "LC_F",        MAIN,        -0.0243f,  -299.094f },    // load cell: slope, offset
  { I2C,       0x2A,       "LC_E",        AUX1,        -0.0242f,  -299.096f },
  { AI,        34,         "FSR",         MEDIAN,       3 },                      // median filter, window 3
};
const size_t deviceList_len = sizeof(deviceList) / sizeof(deviceList[0]);

void setup() {
  exo.setupCommunication("wifi", "SSID", "PASSWORD", "192.168.1.10", 8888[6]);   // or: setupCommunication("usb")
  if (exo.initializeSystem()) {
    exo.setupNode("hub_right_node"[7]);
    exo.setupPublisher("/hub_right"[8], 100[9]);                                  // topic, rate in Hz
    exo.bootSystem();                                                                  // always last
  }
}

void loop() {
  exo.readSensorsData();
  exo.publish(true);                      // true: rebuild the link automatically if the agent restarts
}
```

1. device type: `AI`, `DI` or `I2C`
2. GPIO pin (AI, DI) or 7-bit address (I²C)
3. short label for your reference
4. meaning depends on the type: see the table below
5. misc1 / misc2: meaning depends on the type and driver
6. Wi-Fi network, password, IP address of the host running the agent, UDP port
7. ROS 2 node name of this board (unique per board)
8. topic name: must equal `hubs.<hub>.topic` in `sensors.yaml`
9. publication rate (Hz); also the sampling rate used to design the analog filters

What each field of a device row means

Table S1. Device descriptor. One row of `deviceList[]` declares one device. The same seven fields serve every device type; four of them are read according to the type.

Field	Analog input `AI`	Digital input `DI`	I²C device `I2C`
`IP_type` device type	`AI`	`DI`	`I2C`
`pin_or_address` where it is connected	ADC pin; GPIO 32–39 (ADC1) when the board uses Wi-Fi	GPIO pin	7-bit address of the chip, e.g. `0x4A`
`name` label	Short label for your reference, e.g. `"FSR"`, `"HEEL"`, `"IMU_T"`		
`extra_info1` option	**Filter:** `FILTER_NONE` (default), `MOV_AVG`, `MEDIAN`, `EXPONENTIAL`, `BUTTER2_LPF`, `BUTTER2_HPF`, `BUTTER2_BPF`, `BUTTER2_NOTCH`	**Pin mode:** `NORMAL` (default), `PDOWN` (pull-down), `PUP` (pull-up)	**Bus:** `MAIN` (default): SDA 21, SCL 22 `AUX1`: SDA 19, SCL 18
`dev_misc_data` parameter 1	`MOV_AVG`, `MEDIAN`: window, samples (≤ 50) `EXPONENTIAL`: α, 0–1 (smaller = smoother) `BUTTER2_*`: cut-off or centre frequency, Hz	n/a	**BNO085:** channels 3, 6 or 9 (other values give 3) **NAU7802, AS5600:** not used
`dev_misc_data2` parameter 2	`BUTTER2_*`: Q, required (0.707 = Butterworth) other filters: not used	n/a	n/a
`data_per_device` set by ESA: omit	1	1	set by the driver: NAU7802 1 · AS5600 3 · BNO085 3, 6 or 9
Example row	`{ AI, 34, "FSR", MEDIAN, 3 }`	`{ DI, 26, "HEEL", PUP }`	`{ I2C, 0x4A, "IMU_T", MAIN, 9 }`

Fields are positional; trailing fields may be left out and are then 0, which gives the defaults marked above. n/a: not used.
Butterworth filters are designed for the publication rate given to `setupPublisher()`.
Driver calibration (NAU7802 slope and zero offset, AS5600 direction pin) is not part of the row; it is set in the driver's registration line (Table S2).

The I²C drivers shipped with ESA:

I²C driver	Address	Channels (in order)	misc1	misc2
NAU7802 load-cell amplifier	0x2A	force, N (tension only)	n/a (slope and offset are set in the driver's registration line)	n/a
BNO085 IMU	0x4A, 0x4B	3: roll, pitch, yaw (deg) · 6: + angular velocity x,y,z · 9: + linear acceleration x,y,z	3, 6 or 9	n/a
AS5600 magnetic encoder	0x36	angle (deg), angular speed (deg/s), cumulative position (counts)	n/a (direction pin GPIO 4, set in the driver)	n/a

B. ESA sketch: actuation board

`actuator_board.ino`

```
#include "ESA.h"
SoftExoWalk exo;

DeviceConfig deviceList[] = { };                    // no sensors on this board
const size_t deviceList_len = sizeof(deviceList) / sizeof(deviceList[0]);

void setup() {
  exo.setupCommunication("wifi", "SSID", "PASSWORD", "192.168.1.10", 8888);
  if (exo.initializeSystem()) {
    exo.setupNode("actuator_node");
    //                  command topic              actuator                  out    x1   x2
    exo.setupActuation("/right/hip_ppr_cmd"(1), ACT_FESTO_VEAB_L26(2), 26(3), 17(4), 34(5));
    exo.setupActuation("/left/hip_ppr_cmd",        ACT_FESTO_VEAB_L26,        25,     5,  39);
    exo.bootSystem();
  }
}

void loop() {
  exo.actuate();              // all actuators; exo.actuate("/right/hip_ppr_cmd") drives only that one
}
```

(1) command topic: must equal `output_topic` of a planner in `mplanner.yaml`
(2) identifier of a registered actuator driver (table below)
(3) output: GPIO pin or I²C address
(4) first driver-specific value (omit if unused)
(5) second driver-specific value (omit if unused)

Actuator	out	x1	x2	Command (Float32)
`ACT_FESTO_VEAB_L26` proportional pressure valve	analog output GPIO (25, 26)	exhaust valve GPIO	pressure-feedback ADC GPIO (omit if none)	pressure set-point, kPa
`ACT_SG90_SERVO` servo via PCA9685	PCA9685 I²C address (e.g. 0x40)	PWM channel 0–15	PWM frequency, Hz (50)	angle, 0–180°
`LED_TEST` test output	GPIO	n/a	n/a	> 0 → on, otherwise off

Table S2. Driver registries. Each driver enters itself in a registry through one registration line in its header, before `setup()` runs. ESA then finds the driver for every device without any change to the library.

	I²C sensor registry	Actuator registry
Registration line in the driver header	`REGISTER_I2C_SENSOR(addr, bus, INIT, READ, info1, info2)`	`REGISTER_ACTUATOR(ID, INIT, ACTUATE)`
Key	bus and 7-bit address, e.g. `MAIN` + `0x2A`	identifier of the `ActuatorType` enum in `actuation.h`, e.g. `ACT_FESTO_VEAB_L26`
Selected by	a device row `{ I2C, addr, name, bus, … }`, matched in `initializeSystem()`	`setupActuation(topic, ID, out, x1, x2)`, resolved in `bootSystem()`
Driver receives	`info1`, `info2` of the matching registration line	`out`, `x1`, `x2` of the call (values ≤ 0 arrive as 0)
No matching entry	the device fails; `initializeSystem()` returns false	the actuator is skipped without an error
Capacity	8 entries (`MAX_I2C_SENS`); the shipped lines use all 8	one entry per identifier; up to 5 actuators per board
Entries in v1.0	**NAU7802**: `MAIN 0x2A`, `AUX1 0x2A`; info1 slope, info2 zero offset, calibrated per bus **BNO085**: `MAIN 0x4A`, `MAIN 0x4B`, `AUX1 0x4A`, `AUX1 0x4B`; no info values **AS5600**: `MAIN 0x36`, `AUX1 0x36`; info1 = 4, the direction pin	`ACT_FESTO_VEAB_L26`: proportional pressure valve `ACT_SG90_SERVO`: servo through a PCA9685 `LED_TEST`: test output
Adding a driver	a header–source pair with its registration lines (§2.5)	a header–source pair with its registration line, and the identifier added to `ActuatorType` (§2.6)

An I²C entry is looked up once and then called through a cached pointer; actuator entries are indexed directly by their identifier.

ESA RULES

- **Call order:** `setupCommunication` → `initializeSystem` → `setupNode` → `setupPublisher` / `setupActuation` → `bootSystem`. The analog filters are designed when `setupPublisher` runs, so it must come after `initializeSystem`.
- **USB transport:** micro-ROS uses the serial port, so do not call `Serial.begin` or `Serial.print` in the sketch.
- **Wi-Fi:** analog inputs must use ADC1 pins (GPIO 32–39); ADC2 is unavailable while Wi-Fi is active.
- **I²C buses:** `MAIN` = SDA 21 / SCL 22, `AUX1` = SDA 19 / SCL 18, 400 kHz. A bus–address pair can be used once.
- A board may publish and actuate at once: call both `setupPublisher` and `setupActuation`, and in `loop()` call `readSensorsData()`, `publish(true)` and `actuate()`. Up to 5 actuators per board.
- An analog device whose name starts with `BAT` is treated as a battery monitor (×1.34 divider, clipped at 4.2 V).
- Status LED on GPIO 23 blinks if a ROS entity cannot be created; the watchdog resets a stalled board after 3 s.

C. sensors.yaml: hubs and channel names

One entry per sensing board. List **one name per published channel, in the exact order the board publishes them**: devices in `deviceList` order, each expanded to its channel count. These names are how every algorithm refers to a signal.

`~/exo_stack_resources/config/sensors.yaml`

```
hubs:
  hubr (1):
    topic: "/hub_right" (2)
    sensors: ["thigh_roll", "thigh_pitch", "thigh_yaw", "thigh_omega_x", "thigh_omega_y", "thigh_omega_z", (3)
              "thigh_acc_x", "thigh_acc_y", "thigh_acc_z",
              "shank_roll", "shank_pitch", "shank_yaw", "shank_omega_x", "shank_omega_y", "shank_omega_z",
              "shank_acc_x", "shank_acc_y", "shank_acc_z",
              "force_flexion", "force_extension", "fsr"]
  hubl:
    topic: "/hub_left"
    sensors: [ ... same 21 names ... ]
slop_time: 0.010 (4)
output_topic: "/synced_sensors"
```

1. hub key: used as `[hub, name]` in the other files
2. the board's publisher topic
3. channel names in publication order
4. seconds; messages from different hubs closer than this are combined into one sample

Device row (sketch)	Channels	Slots	Names in sensors.yaml
IMU_T (BNO085, misc1 = 9)	9	0–8	`thigh_roll … thigh_acc_z`
IMU_S (BNO085, misc1 = 9)	9	9–17	`shank_roll … shank_acc_z`
LC_F (NAU7802)	1	18	`force_flexion`
LC_E (NAU7802)	1	19	`force_extension`
FSR (AI)	1	20	`fsr`

CHECK THE CHANNEL COUNT

With the board running, `ros2 topic echo /hub_right --once` shows `range_max` = number of channels and `range_min` = number of devices. The `sensors` list must have exactly `range_max` names. Keep `output_topic: "/synced_sensors"`; the cost node subscribes to that name.

D. gait_events.yaml: event detectors

`~/exo_stack_resources/config/gait_events.yaml`

```
events:
  right_heel_strike (1):
    input_topic: "/synced_sensors"
    sensors: [[hubr, fsr (2)]]
    algorithm: "HeelStrike" (3)
    parameters: [2.0, 0.2 (4)]
    history_size: 5 (5)
    output_topic: "/heel_strike_right" (6)
```

1. event key: becomes the node name; one detector node runs per key
2. `[hub, channel]` pairs passed to the detector, in this order
3. detector plug-in: file and class name in `algo/gait_events/`
4. list handed to the detector; meaning per algorithm (table)
5. sliding-window length in samples (PASE: *Window Size*)
6. topic on which an `Int32` pulse (value 1) is published at each event

Detector	parameters	Fires when
`HeelStrike`	[threshold, fraction]	the signal rises above *threshold* and stays there for *fraction* of the window, after a low window
`ZeroCrossingPosToNeg`	[fraction]	the signal crosses from positive to negative and stays negative for *fraction* of the window
`ZeroCrossingNegToPos`	[fraction]	the reverse crossing

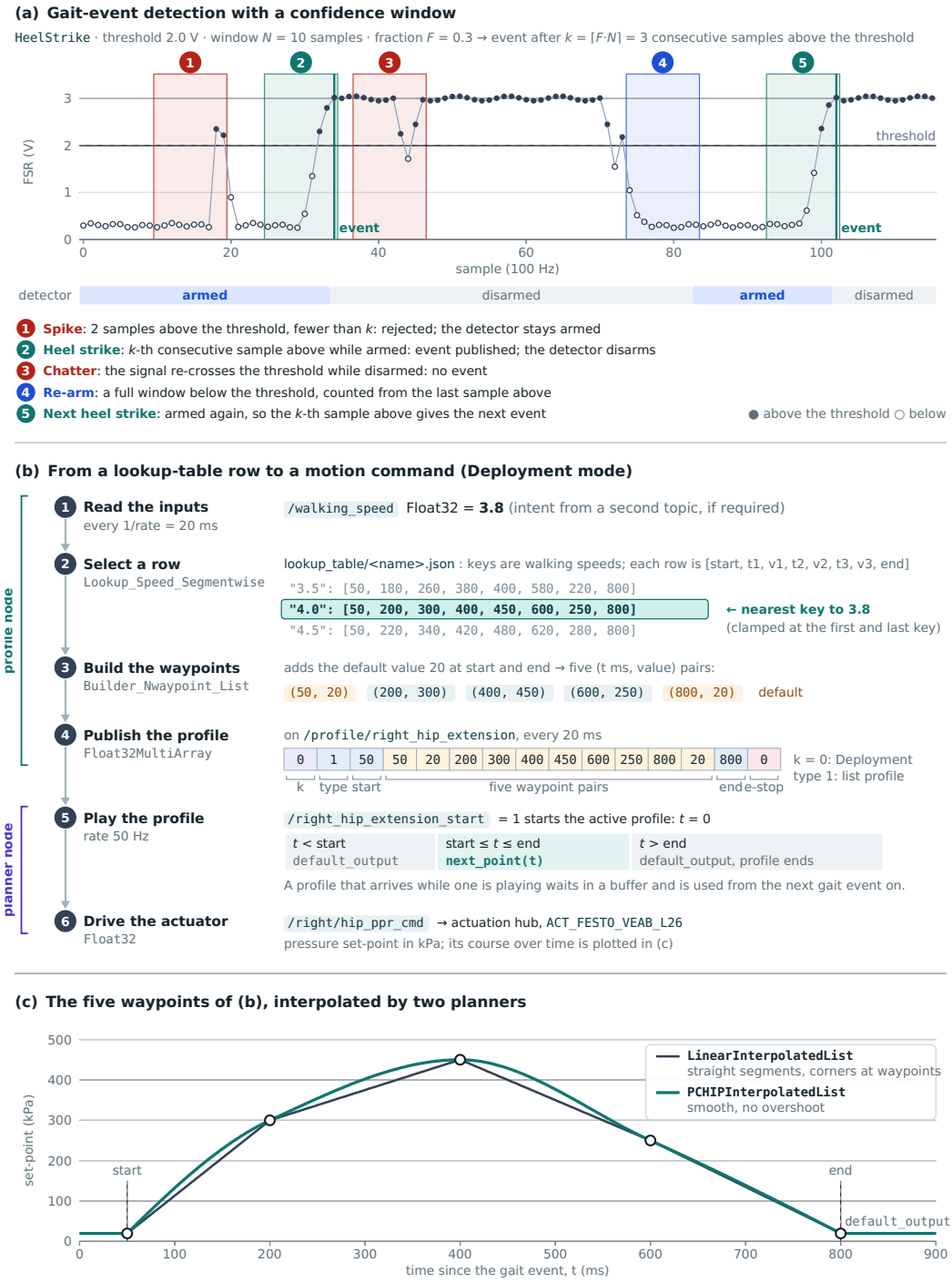


Fig. S1. ExoCore elements, computed with the shipped plug-ins. (a) `HeelStrike` on an FSR signal (window 10 samples, fraction 0.3): an event needs $k = 3$ consecutive samples above the threshold while the detector is armed, and after an event it re-arms only once a full window lies below the threshold, so short spikes and chatter are rejected. The shipped `gait_events.yaml` uses `history_size: 5` and fraction 0.2, for which one sample above the threshold is enough; the zero-crossing detectors apply the same rule to the sign of the signal. (b) One profile node and one motion planner node turn the walking speed into actuator set-points; the lookup table is an example in the row format of `Builder_Nwaypoint_List`. (c) The five waypoint pairs of (b) as interpolated by `LinearInterpolatedList` and `PCHIPInterpolatedList`; the PCHIP curve is the plug-in's interpolation with its inverted length check corrected (Known limitations of v1.0).

E. mplanner.yaml: motion planners

~/exo_stack_resources/config/mplanner.yaml

```yaml
planners:
  right_hip_extension:
    gait_event_topic: "/right_hip_extension_start"
    planner_algorithm: "LinearInterpolatedList"
    profile_type: "List"
    profile_topic: "/profile/right_hip_extension"
    output_topic: "/right/hip_ppr_cmd"
    approach: "time_based"
    rate: 50
    default_output: 20.0
```

1. planner key: node name; referenced by `planner_name` in `profile.yaml`
2. event that (re)starts the profile: an `output_topic` from `gait_events.yaml`
3. planner plug-in in `planner/`; its name ends in `List` or `Function`; interpolation compared in Fig. S1c
4. `List` or `Function`: must match the suffix of the planner and of the builder
5. any unique name; the profile node publishes the profile here
6. `Float32` command to the actuator: same topic as `setupActuation`
7. `time_based`: t = milliseconds since the event · `gait_phase_based`: t = latest value on `gait_phase_topic` (add that key)
8. command publication rate, Hz
9. command when idle, before the profile starts, after it ends, or on emergency stop

Topic (created per planner)	Type	Use
`/<planner>/manual_mode`	Int32	1 = manual mode on, 0 = off (profiles are reset)
`/<planner>/manual_cmd`	Float32	in manual mode the output is `default_output + manual_cmd`
`/<planner>/profile_status`	Float32MultiArray	[profile number, gait-cycle number], at every gait event

F. profile.yaml: where the assistance profile comes from

Each profile feeds one planner. In **Deployment** mode the parameters come from a lookup table indexed by walking speed (and optionally intent); in **HIL** mode they come from an optimizer that is scored by a cost function.

Key	Mode	Meaning
`planner_name`	both	planner key in `mplanner.yaml`
`mode`	both	`Deployment` or `HIL`
`profile_builder_algo`	both	builder in `algo/builder/` (table below); suffix must match the planner
`speed_required`, `speed_topic`	Deployment	use walking speed (Float32 topic) to index the table
`intent_required`, `intent_topic`	Deployment	also index by intent (Float32); intent 0 triggers an emergency stop
`lookup_algo`, `lookup_table`	Deployment	plug-in in `algo/lookup_algo/`; JSON file in `lookup_table/` (name without .json)
`rate`	Deployment	profile republication rate, Hz (default 50)
`hil_algo`	HIL	optimizer in `algo/hil_optimizer/` (upper-case name)
`hil_output_topic`	HIL	topic on which candidates are sent to the profile node (any unique name)
`hil_hyperparams`	HIL	optimizer settings (next table)
`cost_algo`, `cost_params`	HIL	cost plug-in in `algo/cost/`; list passed to its constructor
`cost_topic`	HIL	topic on which [profile number, cost] is returned (any unique name)
`sensors`	HIL	`[hub, channel]` pairs given to the cost function, in this order
`gait_cycles_to_skip`, `_to_consider`	HIL	cycles discarded after each new candidate, then cycles averaged into one cost
`external_sensors_topic`	HIL, optional	Float32MultiArray stream (e.g. metabolic, EMG) used instead of internal sensors
`additional_gait_marker_topic`	HIL, optional	Int32 marker appended as an extra column for the cost function

Optimizer	hil_hyperparams
`OPT_CMA_ES`	`mean` (start point), `bounds` ([[min,max],...]), `sigma`, `maximize`, `max_generations`, `population_size`, `n_max_resampling`
`OPT_BAYESIAN`	`pbounds` ({name: [min,max], ...}; key order = parameter order), `maximize`, `max_iters`, `kind` (`ei`, `ucb`, `poi`), `xi`, `kappa`, `random_state`

Builder	Input values (from lookup table or optimizer)	Shape
`Builder1_List`	[start, peak_t, peak_value, end]	triangle
`Builder2_List`	[start, t1, v1, t2, v2, end]	trapezoid
`Builder_Nwaypoint_List`	[start, t1, v1, ..., tN, vN, end]	any piecewise shape
`Builder1_Function`	[start, c0, c1, ..., cn, end]	polynomial (Function planners); see App. A for v1.0

MATCH THE PARAMETER COUNT

The optimizer must produce exactly as many values as the builder expects, e.g. six for `Builder2_List`: six entries in `mean` and `bounds`, or six keys in `pbounds`. Times are in milliseconds for `time_based` planners and in gait-phase units for `gait_phase_based` ones. The List builders anchor the profile at their own default value (20) at the start and end times; this is independent of the planner's `default_output`.

G. Lookup tables

`lookup_table/speed_list_lookuptable_2.json (speed only)`

```
{ "3.5": [50.0, 200.0, 500.0, 600.0, 500.0, 800.0],          ← speed -> builder inputs
  "4.0": [200.0, 300.0, 100.0, 500.0, 150.0, 800.0] }
```

`lookup_table/speed_intent_list_lookuptable_1.json (intent and speed)`

```
{ "0.0": [500.0, 600.0, 100.0, 800.0],                       ← intent -> builder inputs ...
  "1.0": { "3.5": [500.0, 600.0, 100.0, 800.0],              ← ... or intent -> speed -> inputs
           "4.0": [400.0, 550.0, 150.0, 800.0] } }
```

Fig. S1b follows one row of such a table to the encoded profile. `Lookup_Speed_Segmentwise` uses the nearest speed key (clamped at the ends); `Lookup_Speed_Interpolated` interpolates between keys; `Lookup_Speed_Intent_Segmentwise` reads the two-level table. Write keys exactly as floats (`"1.0"`, not `"1"`): incoming values are " "matched as text.

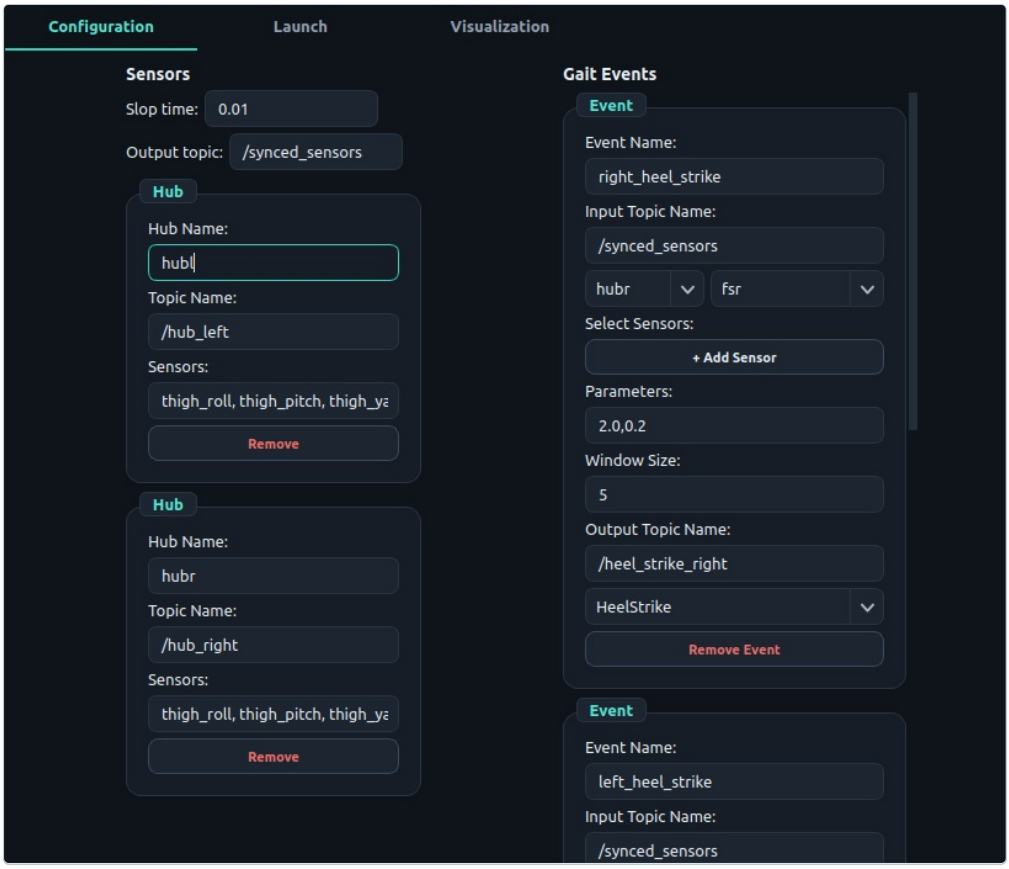
H. Configuring in PASE (Configuration tab)

The four panels edit the four YAML files, left to right. Lists of hubs, channels, events, planners and algorithms are filled from the saved files and from the plug-in folders, so you select rather than type.

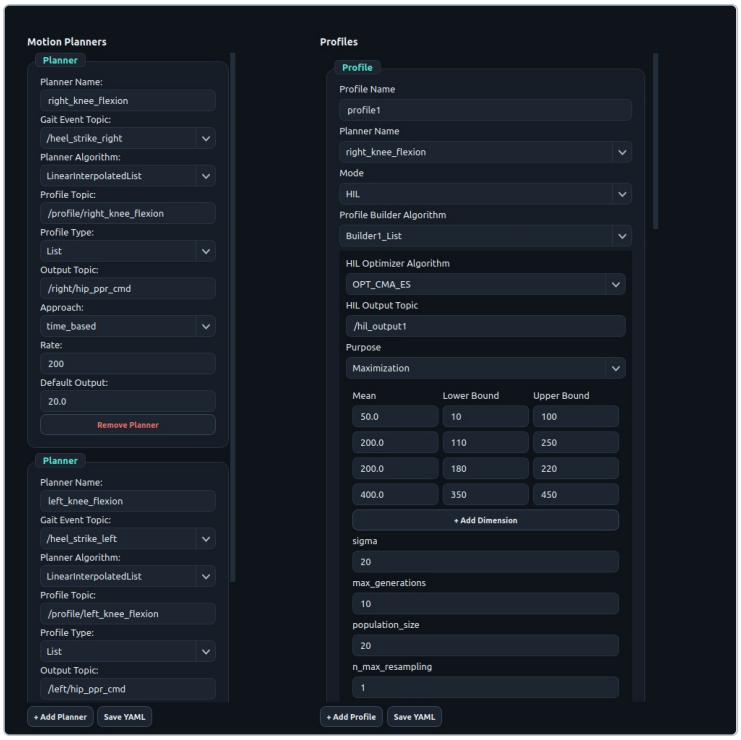
1. **Sensors**: + *Add Hub*; enter *Hub Name*, *Topic Name* and the channel names in *Sensors* (comma-separated, publication order); set *Slop time* and *Output topic*; press **Save YAML**.
2. **Gait Events**: + *Add Event*; set *Event Name*, *Input Topic Name*; choose hub and channel under *Select Sensors* (+ *Add Sensor* for more); pick the algorithm; fill *Parameters* and *Window Size*; set *Output Topic Name*; **Save YAML**.
3. **Motion Planner**: + *Add Planner*; choose the *Gait Event Topic* and *Planner Algorithm* from the lists; set *Profile Type*, *Profile Topic*, *Output Topic*, *Approach* (`time_based` ; phase-based planners are set in the YAML file), *Rate*, *Default Output*; **Save YAML**.
4. **Profile**: + *Add Profile*; choose *Planner Name* and *Mode*. For *Deployment* set the builder, *Speed/Intent Required*, *Lookup Table*, *Lookup Algorithm*, *Rate*. For *HIL* choose the optimizer, add one + *Add Dimension* row (*Mean / Lower Bound / Upper Bound*) per builder input, then under *Define Cost Function* set the cost algorithm, *Cost Params*, *Cost Topic*, *Sensors for Cost* and the gait cycles to skip and consider; **Save YAML**.

SAVE FROM LEFT TO RIGHT

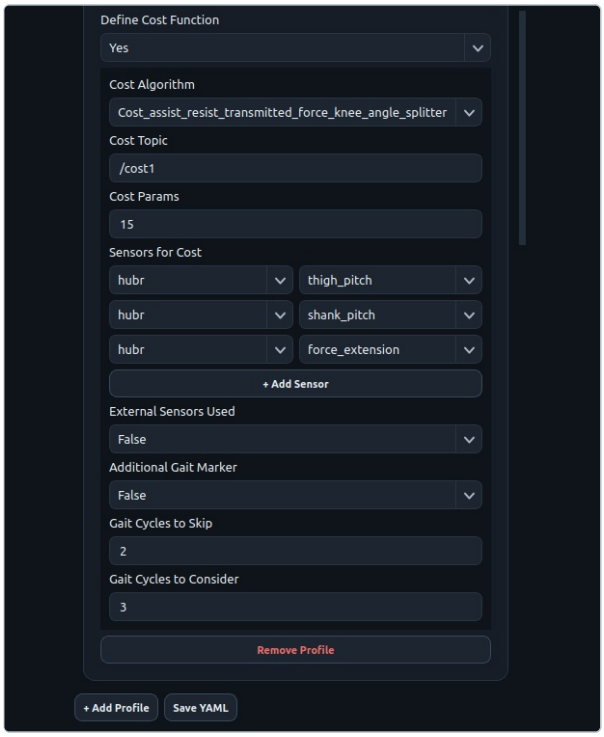
Saving a panel refreshes the lists of the panels to its right. A plug-in file you add to a folder appears in the corresponding list the next time PASE starts.



Screenshot S2.2-1. Sensors and Gait Events panels: the two hubs with their channel names, and the heel-strike detectors, each reading one [hub, channel] pair.



(a) Motion Planners and Profiles



(b) cost function of the profile

Screenshot S2.2-2. (a) Motion Planners and Profiles panels with a HIL profile (optimizer OPT_CMA_ES, one row per optimized dimension); (b) the cost-function block of the same profile.

2.3 Running the stack

The stack runs as seven stages. Start them in the order below, from the PASE **Launch** tab or from terminals; the two are equivalent.

#	Stage	PASE (Launch tab)	Terminal command	Check that it works
1	Hubs (micro-ROS agent)	HUB CONNECTIONS → *Connect Hubs*	Wi-Fi: `ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888` USB: `… serial --dev /dev/ttyUSB0 -b 115200`	power the boards; `ros2 topic list` shows `/hub_right`
2	Synchronizer	SYNCHRONIZER → *Start Sync*	`ros2 run exo_stack sync`	`ros2 topic hz /synced_sensors`
3	Gait detection	GAIT DETECTION → *Start Detection*	`ros2 launch exo_stack gait_event_detector_launch.py`	`ros2 topic echo /heel_strike_right` while walking
4	Motion planning	MOTION PLANNING → *Start Planning*	`ros2 launch exo_stack mplanner_launch.py`	output topics carry `default_output`
5	Profiles	PROFILE MANAGER → *Start Profile*	`ros2 launch exo_stack profile_launch.py`	`ros2 topic echo /profile/right_hip_extension`
6	Cost (HIL)	COST COMPUTATION → *Start Computation*	`ros2 launch exo_stack cost_launch.py`	cost logged after the considered cycles
7	HIL optimizer	HIL OPTIMIZER → *Start Optimization*	`ros2 launch exo_stack hil_launch.py`	candidate #1 logged

START THE OPTIMIZER LAST

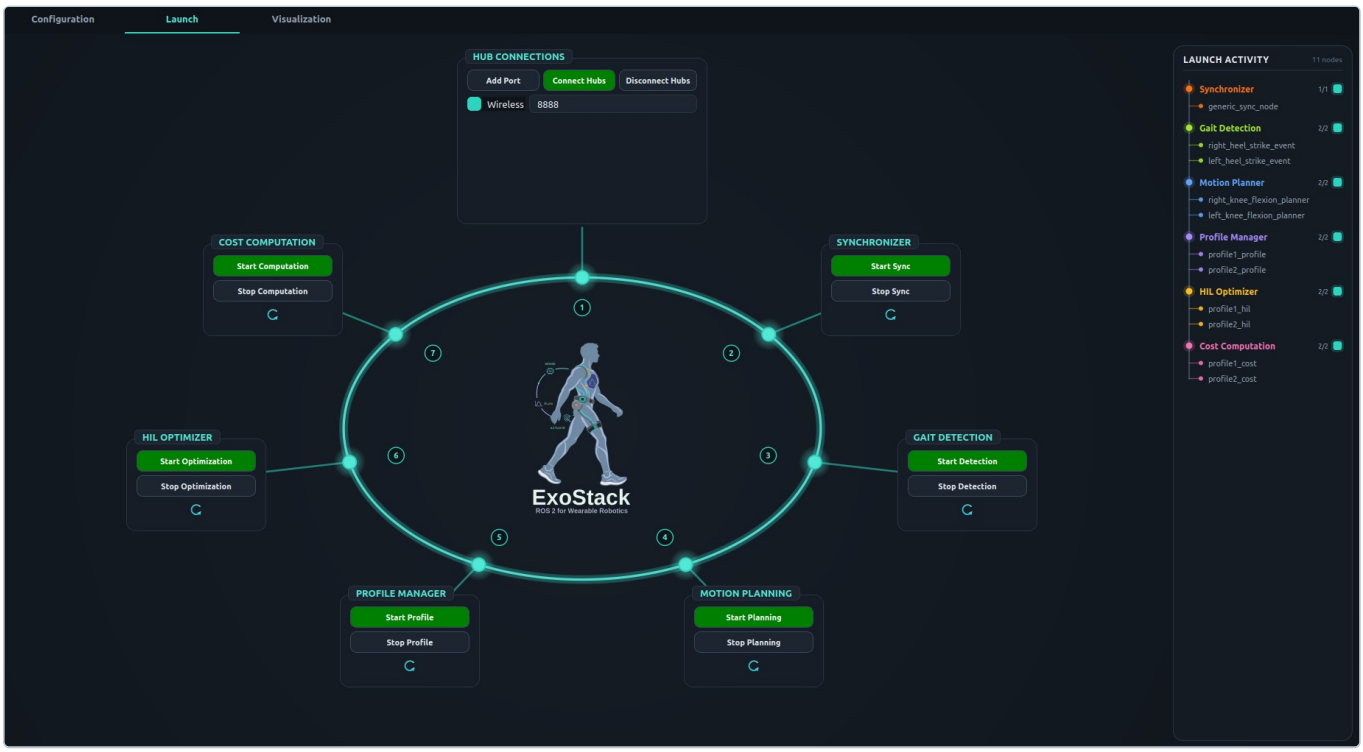
The HIL optimizer sends its first candidate as soon as it starts. Start it only after the profile, planner and cost stages are running, otherwise the first candidate is lost and the optimizer waits indefinitely for its cost. In Deployment mode, stop at stage 5.

- Each launch file starts **one node per entry** of its YAML file, named `<key>_event`, `<key>_planner`, `<key>_profile`, `<key>_cost`, `<key>_hil`.
- To run a single entry while debugging, pass its key: `ros2 run exo_stack gait_event_detector right_heel_strike` (likewise `mplanner`, `actuation_profile`, `cost`, `hil` with a planner or profile key).
- Each stage box has *Start*, *Stop* and a **restart** icon; restart a stage after you edit and save its YAML file. A stage box lights up while its processes run; the activity tree lists every node expected from the YAML files and marks the ones that are alive.
- Stop in reverse order. *Disconnect Hubs* stops the agents.
- HIL results are written when the optimizer finishes, to `~/exo_stack_resources/hil_results/HIL_<optimizer>_<cost>_…_<time>.csv` (one row per candidate: number, parameters, cost).

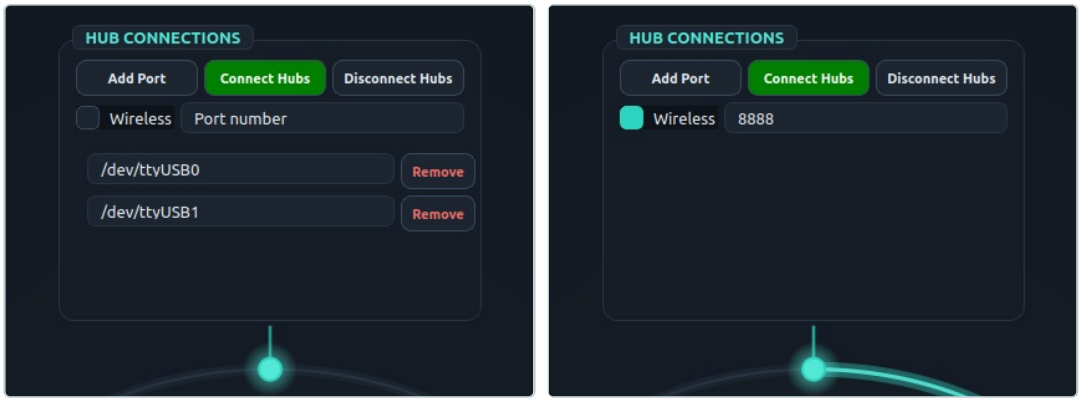
Manual override of a planner

terminal

```
$ ros2 topic pub --once /right_hip_extension/manual_mode std_msgs/msg/Int32 "{data: 1}"     # enter manual mode
$ ros2 topic pub --once /right_hip_extension/manual_cmd  std_msgs/msg/Float32 "{data: 30.0}" # output = default + 30
$ ros2 topic pub --once /right_hip_extension/manual_mode std_msgs/msg/Int32 "{data: 0}"     # back to profiles
```



Screenshot S2.3-1. Launch tab with every stage started: each stage box is lit and the activity tree on the right lists the live nodes.



(a) USB (b) Wi-Fi

Screenshot S2.3-2. HUB CONNECTIONS box: (a) USB, one serial port per hub; (b) Wi-Fi, with the agent on UDP port 8888.

2.4 Visualization panel

The Visualization tab plots any field of any topic live, with a rolling 10-second window. The topic tree is on the left, the plot area on the right.

To …	Do this
update the topic list	press ↻ **Refresh Topics** (after starting new stages)
plot a signal	expand a topic in the tree and **drag a field onto a plot** (“Drop a topic field here”). Several fields can share one plot
plot a sensor by name	expand `/synced_sensors` : it is listed as *hub* → *channel* using `sensors.yaml` , so you never count array indices
add a plot	right-click an empty part of the plot area → **Add Blank Plot**
split the screen	right-click a plot → **Split Vertically** or **Split Horizontally**; repeat for a grid
remove a plot	right-click the plot → **Remove Plot**
show events as markers	right-click the plot → choose the curve → **Event**: every message is drawn as a dashed vertical line. Use it for gait-event topics such as `/heel_strike_right`
return to a line plot	right-click → curve → **Continuous**
change a curve colour	right-click the curve's **legend label** → choose a colour
freeze the display	▶/⏸ pauses and resumes all plots (data keeps arriving)
save the arrangement	💾 **Save Layout** → JSON file with plots, splits, curves, modes and colours
restore it	📂 **Load Layout** → the same JSON file

A USEFUL TUNING LAYOUT

Split into three rows: (1) `/synced_sensors → hubr → fsr` with `/heel_strike_right` in **Event** mode on the same plot, to check detection timing; (2) `/right/hip_ppr_cmd` , the command sent to the valve; (3) `/synced_sensors → hubr → force_extension` , the force actually delivered. Save it once and load it at every session.



Screenshot S2.4-1. Visualization tab: the topic tree on the left, and a plot holding a force channel and a heel-strike topic. The right-click menu changes a curve's colour, sets it to Continuous or Event mode (events appear as dashed vertical lines), splits the plot, or removes it.

2.5 ESA: adding an I²C sensor driver

A new I²C sensor is one header and one source file in `src/sensing/I2C/`. The header ends with a registration line; at program start the driver enters itself in the I²C registry (Table S2) under its bus–address pair, and any device-list row with that pair uses it. No other file of the library is edited.

Item	Convention	Example (shipped)
Location	`ESA/src/sensing/I2C/`	n/a
Files	`<DEV>.h` and `<DEV>.cpp`	`NAU7802.h`, `NAU7802.cpp`
Functions	`<DEV>__INIT_` and `<DEV>__READ_`: two underscores before, one after	`NAU7802__INIT_`, `NAU7802__READ_`
Registration	one `REGISTER_I2C_SENSOR` line per address and bus, in the header	`REGISTER_I2C_SENSOR(0x2A, MAIN, …)`
Vendor library	installed in `~/Arduino/libraries/`	`SparkFun_Qwiic_Scale_NAU7802…`

Template: header

`src/sensing/I2C/MYSENS.h`

```
#pragma once
#include "I2C_Manager.h"
#include "MySensorLib.h"(1)                  // the chip's Arduino library

bool MYSENS(2)__INIT_(I2C_Sens_Ctx* sens);
bool MYSENS(2)__READ_(I2C_Sens_Ctx* sens);

//                address     bus        init function     read function     info1, info2 (optional)
REGISTER_I2C_SENSOR(0x48(3),  MAIN(4),  MYSENS__INIT_,  MYSENS__READ_,  1.0, 0.0(5));
REGISTER_I2C_SENSOR(0x48,  AUX1,  MYSENS__INIT_,  MYSENS__READ_,  1.0, 0.0);    // same chip on the second bus
```

1. header of the vendor library for the chip
2. device name `<DEV>`; use the same name for the file, both functions and in the .cpp
3. the chip's 7-bit I²C address; add one line per address the chip can take
4. `MAIN` (SDA 21 / SCL 22) or `AUX1` (SDA 19 / SCL 18); add a line per bus you wire it to
5. values of `info1`, `info2` handed to the driver for this bus and address (e.g. calibration constants)

Template: source

src/sensing/I2C/MYSENS.cpp

```
#include "MYSENS(1).h"

static constexpr uint8_t N_CH = 2(2);                  // channels this device publishes

bool MYSENS__INIT_(I2C_Sens_Ctx* sens) {
    if (!sens) return false;
    TwoWire& wire = (sens->line == MAIN) ? Wire : Wire1;  // bus chosen in the device list

    MySensor* dev = new MySensor();(3)                    // one driver object per instance
    for (int i = 0; i <= _MAX_SENSOR_INIT_RETRIES; i++) {
        if (dev->begin(sens->addr, wire)(4)) {
            dev->setRate(100);(5)                         // optional chip configuration
            deviceList[sens->idx].data_per_device = N_CH;  // declare the channel count
            sensorIndices += N_CH;                         // reserve the slots in the message
            sens->sens_instance = (void*)dev;              // keep the object for READ_
            return true;
        }
        delay(5);
    }
    return false;                                          // device reported as not ready
}

bool MYSENS__READ_(I2C_Sens_Ctx* sens) {
    if (!sens || !sens->sens_instance) { sensorIndices += N_CH; return false; }
    MySensor* dev = (MySensor*)sens->sens_instance;

    if (!dev->dataReady()(6)) {                            // no new sample this cycle:
        sensorIndices += N_CH;                             // skip the slots, keep the previous values
        return false;
    }
    msg.intensities.data[sensorIndices++] = dev->readX()(7) * sens->info1 + sens->info2;
    msg.intensities.data[sensorIndices++] = dev->readY()(7);
    return true;                                           // exactly N_CH values, in a fixed order
}
```

(1) include your own header

(2) number of values the device publishes every cycle, the length of its slot range

(3) the vendor driver class (or your own register-level code)

(4) the call that starts the chip at `sens->addr` on `wire`; must return true on success

(5) any one-time chip set-up (range, rate, gain); delete if not needed

(6) the call that reports a new sample; if the chip always has one, use `true`

(7) the reads; scale with `info1` / `info2` where calibration applies; one line per channel

Use it

1. Copy both files into `~/Arduino/libraries/ESA/src/sensing/I2C/` and install the vendor library.
2. Add a row to the sketch: `{ I2C, 0x48, "MYS", MAIN },`.
3. Add `N_CH` names to the hub in `sensors.yaml`, at the position of the new row.
4. Compile and upload; `ros2 topic echo <hub topic> --once` shows the new `range_max`.

DRIVER RULES

- Include the header **only** from its own `.cpp`; including it elsewhere registers the driver twice.
- A bus–address pair belongs to one driver. The registry has a fixed capacity, `MAX_I2C_SENS` in `I2C_Manager.h` (8 in v1.0, all taken by the shipped drivers): delete registration lines for addresses you do not wire, or raise the constant.
- `READ_` runs every cycle: no `delay()`, no blocking waits, no `Serial` output.
- Always advance `sensorIndices` by `N_CH`, even when you write nothing, so that every later channel stays in its slot.

2.6 ESA: adding an actuator driver

A new actuator is one header and one source file in `src/actuation/`. The header registers the driver under an **identifier**, added once to the actuator list in `actuation.h`; the sketch selects the driver by that identifier in `setupActuation()`.

Item	Convention	Example (shipped)
Location	`ESA/src/actuation/`	n/a
Files	`ACT_<DEV>.h` and `ACT_<DEV>.cpp`	`ACT_FESTO_VEAB_L26.h/.cpp`
Functions	`<DEV>__INIT_` and `<DEV>__ACTUATE_`	`FESTO_VEAB_L26__INIT_`, `FESTO_VEAB_L26__ACTUATE_`
Identifier	`ACT_<DEV>`: added to the `ActuatorType` list in `actuation.h`, used in `setupActuation()`	`ACT_FESTO_VEAB_L26`

Template: identifier

src/actuation/actuation.h (excerpt)

```
// UPDATE BELOW ENUM FOR NEW ACTUATOR
typedef enum {
    NO_ACT = 0,
    ACT_FESTO_VEAB_L26,
    ACT_SG90_SERVO,
    LED_TEST,
    ACT_MYACT,                    // new identifier, always before ACTUATOR_COUNT
    ACTUATOR_COUNT
} ActuatorType;
```

1. identifier of the new actuator; the same word is used in its registration line and in the sketch

Template: header

src/actuation/ACT_MYACT.h

```
#pragma once
#include "actuation.h"

void MYACT__INIT_(ActInfo* sub);
void MYACT__ACTUATE_(ActInfo* sub);

REGISTER_ACTUATOR(ACT_MYACT, MYACT__INIT_, MYACT__ACTUATE_)
```

1. driver name `<DEV>`, used for both functions
2. the identifier added to `actuation.h`; use it in `setupActuation()`

Template: source

src/actuation/ACT_MYACT.cpp

```
#include "ACT_MYACT.h"

void MYACT__INIT_(ActInfo* sub) {                // once, during bootSystem()
    if (!sub) return;
    pinMode(sub->out_gpio_i2caddr, OUTPUT);      // 'out' argument of setupActuation()
    if (sub->extra_info1 > 0) pinMode(sub->extra_info1, OUTPUT);   // 'x1', optional
}

void MYACT__ACTUATE_(ActInfo* sub) {             // every loop, from actuate()
    if (!sub) return;
    float cmd = sub->val;                           // latest command (0 before the first message)
    int duty = constrain((int)(cmd * 2.55f), 0, 255);   // convert the physical set-point
    analogWrite(sub->out_gpio_i2caddr, duty);          // drive the hardware
}
```

1. include your own header
2. one-time set-up of the output (pin mode, I²C driver object, PWM frequency …)
3. use `extra_info1` / `extra_info2` (the x1, x2 arguments) for any extra pins or settings; omitted arguments arrive as 0
4. the conversion from the commanded physical quantity (kPa, degrees, %) to the hardware unit
5. the hardware call (`analogWrite`, `digitalWrite`, an I²C driver call …)

ActInfo field	Type	Comes from
`val`	float	latest `Float32` on the command topic
`out_gpio_i2caddr`	int8	`out` argument of `setupActuation()`
`extra_info1`, `extra_info2`	int8	`x1`, `x2` arguments (−128…127; 0 if omitted)
`topicName`	const char*	command topic

Optional: smooth an analog feedback signal

A driver can filter a feedback input with the library's software filters, as the pneumatic valve driver does with its pressure sensor. Register the pin in `INIT_`, read it in `ACTUATE_`:

feedback filter

```
// in MYACT__INIT_ :
act_fb_filter_config[_act_fb_idx].filter_type = 1[1];      // 1 = moving average (filter number = extra_info1 - 8)
act_fb_filter_config[_act_fb_idx].param1      = 10[2];     // window
act_fb_filter_config[_act_fb_idx].gpio        = sub->extra_info2;
_act_fb_idx++;

// in MYACT__ACTUATE_ :
float fb_volts = _act_fb_filt.analog_filter(sub->extra_info2);
```

1. filter: 1 moving average, 2 median, 3 exponential, 4–7 second-order Butterworth LPF/HPF/BPF/notch
2. first filter parameter (window, α or cut-off Hz); `param2` holds Q for Butterworth

Use it

1. Add `ACT_MYACT,` to the `ActuatorType` list in `~/Arduino/libraries/ESA/src/actuation/actuation.h`, before `ACTUATOR_COUNT`.
2. Copy both files into `~/Arduino/libraries/ESA/src/actuation/`.
3. In the actuation sketch: `exo.setupActuation("/my/cmd", ACT_MYACT, out, x1, x2);`
4. Set the same topic as `output_topic` of a planner in `mplanner.yaml`.
5. Compile and upload; test with `ros2 topic pub --once /my/cmd std_msgs/msg/Float32 "{data: 50.0}"`.

DRIVER RULES

- Include the header **only** from its own `.cpp`.
- `ACTUATE_` runs every loop for every actuator: keep it short and non-blocking.
- Variables declared at file scope are shared by all instances of the driver. If a board may run several, keep per-instance state in a small table keyed by `sub->out_gpio_i2caddr`.
- Choose the zero command to be a safe state of the hardware: the driver receives 0 until the first message arrives.

2.7 ExoCore: adding algorithms

Every algorithm in ExoCore is a plug-in: a Python file holding one class, placed in a folder of `~/exo_stack_resources` and named in a YAML file. The node imports it by that name when it starts. Nothing is installed or rebuilt, and PASE lists the new file in its drop-down the next time it starts.

Plug-in	Folder (in ~/exo_stack_resources)	Class name	Constructor	Method called	Named in
Gait-event detector	`algo/gait_events/`	= file name	`(window_size, parameters)`	`detect(values) → 1 | 0`	`gait_events.yaml → algorithm`
Profile builder	`algo/builder/`	= file name, ends in `_List` or `_Function`	`()`	`buildprofile(values) → list`	`profile.yaml → profile_builder_algo`
Lookup algorithm	`algo/lookup_algo/`	= file name	`(json_path)`	`readlookuptable(args) → list`	`profile.yaml → lookup_algo`
HIL optimizer	`algo/hil_optimizer/`	= file name, **UPPER CASE**	`(**hil_hyperparams)`	`get_next_candidate()`, `submit_result()`, `get_best_so_far()`, `is_done()`	`profile.yaml → hil_algo`
Cost function	`algo/cost/`	= file name	`(*cost_params)`	`compute(data) → float`	`profile.yaml → cost_algo`
Planner	`planner/`	= file name, ends in `List` or `Function`	`()`	`next_point(t) → float`; reads `self.profile`	`mplanner.yaml → planner_algorithm`

THE THREE NAMING RULES

1. The class name is identical to the file name without `.py`.
2. Builders end in `_List` or `_Function`, planners in `List` or `Function`; a builder and the planner it feeds must end in the same word (and match `profile_type`).
3. Optimizer files and classes are written in upper case, the node converts the configured name to upper case.

A. Gait-event detector

`algo/gait_events/RisingEdge.py`

```
class RisingEdge(1):

    def __init__(self, window_size=10, parameters=None):
        self.window_size = window_size                    # history_size in gait_events.yaml
        p = parameters or []                              # parameters list in gait_events.yaml
        self.threshold(2) = p[0] if len(p) > 0 else 0.5(3)
        self.window = []

    def detect(self, values):
        # values: one float per [hub, channel] listed under 'sensors', same order
        self.window.append(values[0])
        if len(self.window) > self.window_size:
            self.window.pop(0)
        if len(self.window) < 2:
            return 0
        fired = self.window[-2] <= self.threshold < self.window[-1](4)
        return 1 if fired else 0                          # 1 = publish one event pulse
```

(1) class name = file name; write it under `algorithm`

(2) your parameter names …

(3) … and their defaults, read from `parameters` in order

(4) the detection rule, evaluated once per synchronized sample

B. Profile builder

algo/builder/Ramp_List.py

```
class Ramp(1)_List:

    def __init__(self):
        self.default_value = 20.0                          # level at the start and end of the profile

    def buildprofile(self, values):
        # values: list from the lookup table (Deployment) or the optimizer (HIL)
        if len(values) != 4(2):
            raise ValueError("expected [start, rise_end, peak, end]")
        start, rise_end, peak, end = values(3)
        points = [(start, self.default_value), (rise_end, peak), (end, self.default_value)(4)]
        profile = [1.0, start]                              # 1.0 = List profile, then the start time
        for t, v in points:
            profile += [t, v]                               # waypoints, times strictly increasing
        profile.append(end)                                 # end time
        return profile
```

(1) name ending in `_List` (waypoints) or `_Function` (coefficients)

(2) number of input values: the optimizer dimension or the length of each lookup-table row

(3) the meaning of each input value

(4) the waypoints (time or gait phase, value). A `_Function` builder returns `[2.0, start, c0, c1, …, cn, end]` instead

C. Lookup algorithm

algo/lookup_algo/Lookup_Speed_Nearest.py

```
import json

class Lookup_Speed_Nearest(1):

    def __init__(self, json_path):                          # file named by lookup_table in profile.yaml
        with open(json_path) as f:
            self.table = json.load(f)
        self.keys = sorted(float(k) for k in self.table)

    def readlookuptable(self, args):
        # args: [speed], or [speed, intent] when intent_required is true
        x = float(args[0])
        key = min(self.keys, key=lambda k: abs(k - x))(2)
        return self.table[str(key)]                         # list of builder input values
```

(1) class name = file name

(2) the selection rule: nearest key here; interpolate between keys, or index by intent first

D. HIL optimizer

algo/hil_optimizer/OPT_RANDOM.py

```
import random

class OPT_RANDOM(1):

    def __init__(self, bounds=None, max_iters=20, maximize=True(2)):  # = keys of hil_hyperparams
        self.bounds, self.max_iters, self.maximize = bounds, max_iters, maximize
        self.n = 0
        self.best_point, self.best_score = None, None

    def get_next_candidate(self):
        # one value per builder input
        return [lo + (hi - lo) * random.random() for lo, hi in self.bounds(3)]

    def submit_result(self, candidate, cost):
        # averaged cost of the candidate just evaluated
        self.n += 1
        if self.best_score is None or (cost > self.best_score) == self.maximize:
            self.best_point, self.best_score = candidate, cost
        pass(4)                                          # update your model here

    def get_best_so_far(self):
        return self.best_point, self.best_score

    def is_done(self):                                     # True: results are saved and the node stops
        return self.n >= self.max_iters
```

(1) upper-case name = file name

(2) constructor arguments; each key under `hil_hyperparams` must appear here

(3) how the next candidate is proposed

(4) how a new (candidate, cost) pair updates your optimizer

The node waits for the cost of each candidate before asking for the next one, and discards any cost carrying a different candidate number, so `submit_result` always receives the cost of the candidate it proposed.

E. Cost function

algo/cost/PeakForceTracking.py

```
import numpy as np

class PeakForceTracking(1):

    def __init__(self, target_force(2)):                  # cost_params in profile.yaml, in order
        self.target = target_force

    def compute(self, data):
        # data: NumPy array, one row per sample of ONE gait cycle,
        #       one column per [hub, channel] under 'sensors' in profile.yaml
        force = data[:, 2(3)]
        return -abs(float(np.max(force)) - self.target)(4)
```

(1) class name = file name

(2) constructor arguments, filled positionally from `cost_params`

(3) column of the signal you need (order of `sensors`; a gait-marker column is appended last if configured)

(4) the per-cycle cost; the node averages it over `gait_cycles_to_consider` cycles

F. Planner

planner/NearestList.py

```
class Nearest(1)List:

    def __init__(self):
        self.profile = None
        # set by the planner node before every call:
        #   List planners:     [[t1, v1], [t2, v2], ...]
        #   Function planners: [c0, c1, ..., cn]

    def next_point(self, t):
        # t: ms since the gait event (time_based) or gait phase (gait_phase_based)
        pts = self.profile
        if t <= pts[0][0]:
            return pts[0][1]
        if t >= pts[-1][0]:
            return pts[-1][1]
        for (t0, v0), (t1, v1) in zip(pts, pts[1:]):
            if t0 <= t <= t1:
                return v0 if (t - t0) < (t1 - t) else v1(2)
```

(1) name ending in `List` or `Function`

(2) the interpolation rule: nearest waypoint here; linear, PCHIP, or a polynomial for Function planners

TRY A PLUG-IN WITHOUT THE ROBOT

Every plug-in is plain Python and can be imported and called in an interpreter before it is used in the stack, e.g. `from Ramp_List import Ramp_List; Ramp_List().buildprofile([0, 150, 60, 400])`. After saving the file, restart the stage that uses it (restart icon in the Launch tab).

3 Benchmarking

How regularly the hubs publish and how fast the motion planner runs on three embedded host computers, and the set-up used to measure it.

Test set-up

Item	Set-up
Host computers	Jetson AGX Orin, Jetson Orin Nano, LattePanda Delta; ROS 2 Humble, ExoStack v1.0
Sensing hub	ESP32 running ESA; ten devices per hub: one I²C sensor on `MAIN`, one on `AUX1`, and eight analog inputs
Hub publication	target rate 100 Hz (`setupPublisher(topic, 100)`)
Experiment 1: regularity	up to eight such hubs on each host computer, synchronized; observed publication rate
Experiment 2: loop rate and latency	two such hubs on each host computer; motion-planner target rates of 200, 300, 400 and 500 Hz; achieved planner rate and end-to-end latency from a sensing-hub input to an actuation-hub output

Publication regularity

On each of the three computers, with up to eight hubs publishing at a target of 100 Hz, the observed rate was **99.5 ± 0.018 Hz** (mean ± SD). This is a mean period of 10.05 ms whose spread, $\sigma_T \approx \sigma_f/f^2$, corresponds to a jitter of about **1.8 µs**: the hubs keep their rate as their number grows.

Planner loop rate and end-to-end latency

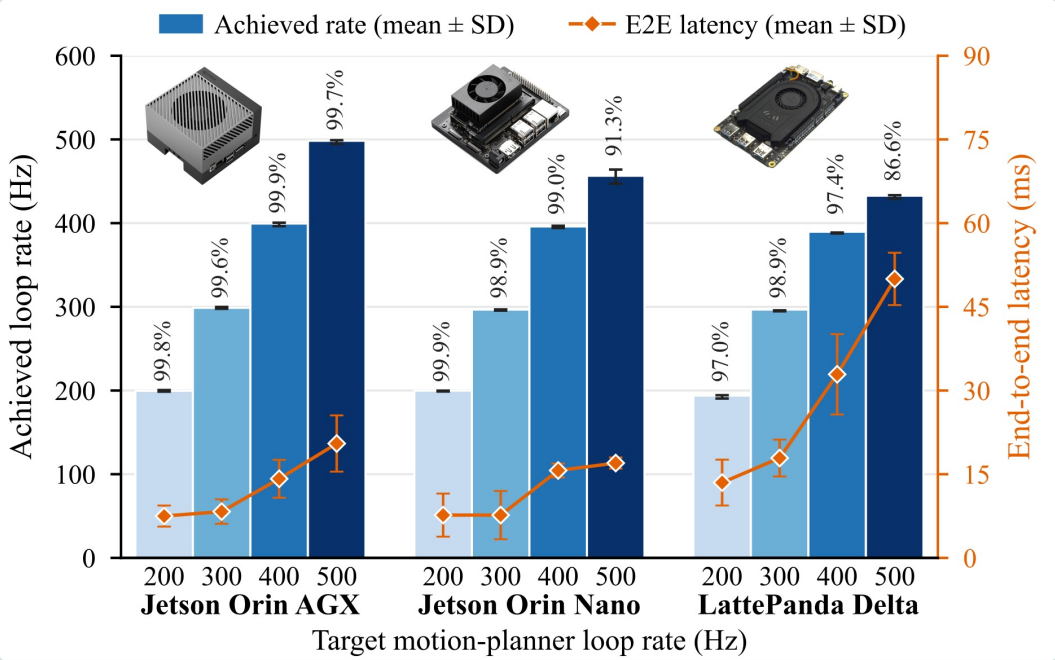


Fig. S2. Benchmarks on three embedded computers. Bars: motion-planner loop rate achieved at each target rate, with the achieved share of the target above each bar. Markers: end-to-end latency from a sensing-hub input to an actuation-hub output. Both are mean ± SD.

Target rate (Hz)	Jetson AGX Orin	Jetson Orin Nano	LattePanda Delta
200	99.8 % · 7.5 ms	99.9 % · 7.4 ms	97.0 % · 13.6 ms
300	99.6 % · 8.4 ms	98.9 % · 7.8 ms	98.9 % · 17.8 ms
400	99.9 % · 14.0 ms	99.0 % · 15.5 ms	97.4 % · 32.9 ms
500	99.7 % · 20.6 ms	91.3 % · 17.2 ms	86.6 % · 50.0 ms

Each cell: share of the target rate achieved by the motion planner · mean end-to-end latency. Shares as printed in the figure; latency means read from the plotted markers (±0.3 ms).

The Jetson AGX Orin achieved 99.6–99.9 % of every target with mean latencies of 8–21 ms. The Jetson Orin Nano kept at least 98.9 % up to 400 Hz (7–16 ms) and 91.3 % at 500 Hz. The LattePanda Delta achieved 97.0–98.9 % up to 400 Hz and 86.6 % at 500 Hz, where its latency rose to about 50 ms. At 200 and 300 Hz all three kept at least 97 % of the target with mean latencies below 18 ms, so a planner rate in this range leaves margin on every host tested.

4 Validated exosuits

The configurations of the two exosuits on which ExoStack was validated.

ExoStack was validated on two pneumatic exosuits built at CSIR-CMERI. Both are actuated by pSTAMs, pneumatic silicone tube artificial muscles, through proportional pressure regulators, and both run the same ESA library and ExoCore nodes: they differ only in their device lists, YAML files and plug-ins. Fig. 5 of the paper shows both in use, and SoftEXOwalk is described in full in its own article.

Configurations

	SoftEXOwalk (knee)	Hip exosuit
Assistance	bilateral knee flexion	bilateral hip flexion and extension
Actuators	one sensorized pSTAM across each knee, on the posterior side	four pSTAMs in a backpack, with tendons to the thighs and a speed-enhancement mechanism that makes them fast enough for walking
Sensing hubs	two, one per leg, over USB. Each reads an insole FSR (analog), two BNO085 IMUs (I²C), a load cell for validation (analog) and a silicone and carbon-black strain sensor on the pSTAM bladder (analog)	two on the thighs, over Wi-Fi, each reading an insole FSR (analog), two BNO085 IMUs (I²C) and two load cells through NAU7802 amplifiers (I²C); two in the backpack, over USB, each reading two encoders (I²C), two muscle strain sensors and two regulator pressures (analog), one of them also a BNO085 IMU (I²C)
Actuation hubs	two, over USB, each driving one proportional pressure regulator and reading back its pressure (analog)	two, over USB, each driving two proportional pressure regulators, one for flexion and one for extension, and four solenoid valves that assist the exhaust of the pSTAMs
Rates	hubs publish at 100 Hz; actuation loop at 100 Hz	hubs publish at 100 Hz; actuation loop at 100 Hz
Gait events	heel strike of each leg, from the FSR (`HeelStrike`)	terminal flexion and terminal extension of each leg, from zero crossings of the IMU angular velocity (`ZeroCrossingPosToNeg` , `ZeroCrossingNegToPos`)
Motion planners	one per leg, time-based, 100 Hz	two per leg, one for flexion and one for extension, time-based, 100 Hz
Profile	trapezoidal	trapezoidal
Deployment	speed lookup table, 3.5 to 5.5 km/h in steps of 0.5 km/h	speed lookup table, 3.5 to 5.5 km/h in steps of 0.5 km/h
HIL parameters	two: the amplitude and the onset delay after heel strike	four: delay, rise time, hold time and fall time, the delay counted from the event that initiates the profile (terminal extension for flexion assistance, terminal flexion for extension)
Optimizer	Bayesian (`OPT_BAYESIAN`)	CMA-ES (`OPT_CMA_ES`)
Cost	assistive and resistive force estimated by the sensorized pSTAM, split by knee angle (`Cost_assist_resist_transmitted_force_knee_angle_splitter` , shipped)	assistive and resistive force, split by the flexion and extension phases of the hip (custom cost plug-in)
Objective	maximize the peak assistive force	maximize the ratio of peak assistive to peak resistive force
Constraint	peak resistive force below 15 N	peak resistive force below 15 N
Personalized for	each speed and each leg	each speed and each leg, for flexion and for extension

The two exosuits in ExoStack terms

Each row of the table becomes entries in the files of §2.2, and each entry one node. The instance counts of the overview figures are therefore:

Instances	SoftEXOwalk	Hip exosuit
Sensing hubs (×N)	2	4
Actuation hubs (×M)	2	2
Gait-event detectors (×E)	2	4
Planners and profiles (×P)	2	4
Cost evaluators and HIL optimizers (×C, ×H)	2	4

In Deployment, the profile nodes pick the trapezoidal profile for the measured walking speed from the lookup table; in HIL, one optimizer per profile tunes its parameters against the cost above, separately for each speed and leg, and the tuned values become the table rows used afterwards.

A Troubleshooting and message reference

Troubleshooting

Symptom	Likely cause	Remedy
Status LED on the board blinks rapidly	a ROS entity could not be created	start the agent; check IP, port and topic names
Board restarts every few seconds	watchdog: `loop()` blocked	remove delays and blocking calls from the sketch and drivers
USB link drops or never connects	the sketch writes to `Serial`	remove all `Serial` calls when using the USB transport
Analog channel stuck at 0 or full scale with Wi-Fi	pin on ADC2	move it to GPIO 32–39
New I²C device not found	duplicate bus–address pair, registry full, or header included elsewhere	see the rules in §2.5
Duplicate-definition errors when compiling	two copies of the ESA library	keep one folder in `~/Arduino/libraries`
`/synced_sensors` silent	a configured hub is not publishing, or a topic name differs	every hub in `sensors.yaml` must be running
Warning *Expected N intensities, got M*	channel list and board disagree	align `sensors.yaml` with the device list (§2.2-C)
FileNotFoundError … exo_stack_resources	folder not in the home directory	move it to `~/exo_stack_resources`
module has no attribute …	class name differs from file name	rename one of them (§2.7)
Planner warns of a profile-type mismatch	suffixes of builder, planner and `profile_type` differ	use List everywhere or Function everywhere
HIL stops after candidate #1	optimizer started before profile and cost stages; or `cost_topic` differs	start HIL last (§2.3)
Profile number mismatch … Ignoring	a late cost from the previous candidate	normal once after each change
PASE cannot start the agent	agent workspace not sourced	add it to `~/.bashrc` (§2.1-E)

ROS 2 and micro-ROS interfaces

Table S3 lists the topic of every ESA hub and ExoCore node and what each element of its message means.

Table S3. ROS 2 and micro-ROS interfaces. Every topic that connects the ESA hubs and the ExoCore nodes, grouped by the node that publishes it. Names in angle brackets are set in the file or call shown under them; all types are standard ROS 2 messages.

Topic	Type	From → to	Content of the message
ESA sensing hub: micro-ROS node, one per board (×N)			
`/<hub topic>` setupPublisher(); sensors.yaml	`sensor_msgs/ LaserScan`	hub → synchronizer	`intensities[i]`: channel i, in device-list order: AI in volts (0–3.3), DI 0 or 1, I²C in the driver's units; `range_min`: number of devices; `range_max`: number of channels; `header.stamp`: hub time since the last `/sync_trigger`. Other fields unused. Sent at the `setupPublisher()` rate.
`/sync_trigger` fixed name	`std_msgs/ Int32`	synchronizer → every hub	`data` = 0, sent at 5 Hz in trigger mode; on reception a hub stores `millis()` as its time offset.
ESA actuation hub: micro-ROS node (×M)			
`/<command topic>` setupActuation(); output_topic in mplanner.yaml	`std_msgs/ Float32`	planner → actuation hub	`data`: set-point in the driver's unit (kPa for `ACT_FESTO_VEAB_L26`, degrees for `ACT_SG90_SERVO`). Kept as the actuator's latest command; 0 until the first message.
Synchronizer: sync (×1)			
`/synced_sensors` output_topic in sensors.yaml	`std_msgs/ Float32MultiArray`	synchronizer → detectors, cost evaluators	`data`: the channels of every hub, concatenated in `sensors.yaml` order (hub by hub, channel by channel), for each set of hub messages whose stamps agree within `slop_time`. No time stamp.
Gait-event detector: gait_event_detector (×E)			
`/<event topic>` output_topic in gait_events.yaml	`std_msgs/ Int32`	detector → planner	`data` = 1 at each detected event; nothing is sent between events.
External estimators and sensors: your own nodes, optional			
`/<speed_topic>` profile.yaml	`std_msgs/ Float32`	estimator → profile node	walking speed, in the unit of the lookup-table keys.
`/<intent_topic>` profile.yaml	`std_msgs/ Float32`	estimator → profile node	locomotion intent; 0 (standing) raises the profile's e-stop flag.
`/<gait_phase_topic>` mplanner.yaml	`std_msgs/ Float32`	estimator → planner	gait phase, used as the time argument t by `gait_phase_based` planners.
`/<external_sensors_topic>` cost block of profile.yaml	`std_msgs/ Float32MultiArray`	sensor → cost evaluator	any vector, e.g. metabolic rate or EMG, handed to the cost plug-in with the synchronized channels.
`/<additional_gait_marker_topic>` cost block of profile.yaml	`std_msgs/ Int32`	your node → cost evaluator	marker value appended to the next cost sample, e.g. a flexion–extension split.
Profile node: actuation_profile (×P)			
`/<profile_topic>` profile_topic of the planner entry	`std_msgs/ Float32MultiArray`	profile node → planner	`[k, type, start, body…, end, stop]`. `k`: candidate number, 0 in Deployment; `type`: 1 = list, 2 = coefficients; `start`, `end`: profile window, in units of t; `body`: (t, value) pairs or coefficients; `stop`: 1 = emergency stop. Sent every 1/rate in Deployment, once per candidate in HIL.
Motion planner: mplanner (×P)			
`/<output_topic>` mplanner.yaml	`std_msgs/ Float32`	planner → actuation hub or any ROS 2 driver	set-point at `rate`: `next_point(t)` inside the profile window; `default_output` outside it or on e-stop; the manual command in manual mode.
`/<planner>/profile_status` planner name	`std_msgs/ Float32MultiArray`	planner → cost evaluator	`[k, cycle]` at each gait event: the candidate being played and its gait-cycle count (0 in Deployment).
`/<planner>/manual_mode` planner name	`std_msgs/ Int32`	PASE or you → planner	1 = manual override on, 0 = back to profiles.
`/<planner>/manual_cmd` planner name	`std_msgs/ Float32`	PASE or you → planner	set-point sent while manual mode is on.
Cost evaluator: cost (×C, one per profile with a cost)			
`/<cost_topic>` profile.yaml	`std_msgs/ Float32MultiArray`	cost evaluator → HIL optimizer	`[k, cost]`: mean of the per-cycle costs over `gait_cycles_to_consider` cycles, after skipping `gait_cycles_to_skip`; `k` = 0 in Deployment.
HIL optimizer: hil (×H, one per profile in HIL mode)			
`/<hil_output_topic>` profile.yaml	`std_msgs/ Float32MultiArray`	optimizer → profile node	`[k, p1 … pn]`: candidate number and one parameter per optimized dimension; a new candidate follows each cost it accepts. On stop, every round and the best candidate are written to `~/exo_stack_resources/hil_results/`.

All ExoCore topics use best-effort reliability, and the hubs publish and subscribe best-effort over micro-ROS.

`k` ties the HIL messages together: the cost evaluator counts only cycles of the candidate in `profile_status`, and the optimizer accepts only a cost carrying the `k` it proposed.

ExoStack uses standard message types only, so no custom message package has to be built for micro-ROS.

KNOWN LIMITATIONS OF v1.0

- Device-list values `dev_misc_data` and `dev_misc_data2` reach the I²C drivers that read them (the BNO085 channel count); the NAU7802 calibration and the AS5600 direction pin come from the drivers' registration lines.
- Function profiles (`Builder1_Function` with `PolynomialFunction` or `CubicPolynomialFunction`) and `PCHIPInterpolatedList` do not run in v1.0; use a `_List` builder with `LinearInterpolatedList` .
- PASE writes the approach `gait_phase` , which the planner does not recognise, and adds `init_points` and `gp_params` to Bayesian profiles, which `OPT_BAYESIAN` does not accept: configure phase-based planners and Bayesian profiles in the YAML files.
- The I²C registry holds 8 entries, all used by the shipped drivers (§2.5).

Corrections are planned for the next release.