IoTeaPot: A Backend-Agnostic Framework for Portable WSN Experimentation

Jeremy Dubrulle
ETRO.RDI, Vrije Universiteit Brussel, Brussels, Belgium
Dept. of Computer Science, UMONS, Mons, Belgium
jeremy.dubrulle@vub.be

Kris Steenhaut
ETRO.RDI, Depts. ETRO and INDI
Vrije Universiteit Brussel, Brussels, Belgium
kris.steenhaut@vub.be

An Braeken
ETRO.RDI, Dept. INDI
Vrije Universiteit Brussel, Brussels, Belgium
an.braeken@vub.be

Bruno Quoitin
Dept. of Computer Science
University of Mons (UMONS), Mons, Belgium
bruno.quoitin@umons.ac.be

***Abstract*—Most Wireless Sensor Network (WSN) studies report results from a single execution platform (a simulator, an emulator, a public testbed, or a small hardware deployment) even though cross-platform validation is widely seen as necessary for credible claims. The barrier is engineering, not science: each backend describes topology, captures logs, schedules events, and reports timestamps in its own way, so moving an experiment between them is costly. We present IoTeaPot, a framework that separates experiment specification from execution. An experiment (topology, firmware, radio medium, event schedule, and observation contract) is described once and translated, unchanged, into different backends such as the Cooja simulator, the Renode emulator, the FIT IoT-LAB testbed, or locally-connected hardware. A generic model captures what these four categories share, realised by a layered architecture that also handles three embedded operating systems and radio capture through one specification surface. One specification currently drives sixteen board/backend/OS combinations unchanged, and a support-level contract makes explicit which backend capabilities cannot be unified.**

***Index Terms*—WSN experimentation, portability, multi-platform evaluation, simulation, emulation, testbed, Contiki-NG, RIOT, Cooja, Renode, FIT IoT-LAB.**

I. Introduction

A typical empirical paper in the Wireless Sensor Network (WSN) literature reports results from a single execution platform: a simulator such as Cooja [1], an emulator such as Renode [2], a public testbed such as FIT IoT-LAB [3], or a small in-house hardware deployment. Each category carries biases the community has documented for over a decade, trading fidelity against the control, scale, and speed an experiment can achieve: *simulators* abstract away silicon-level and real-RF effects; *emulators* execute firmware on a model of the silicon but use a synthetic radio medium; *public testbeds* deploy hundreds of real nodes with realistic RF propagation but constrain topology and instrumentation; *local hardware* gives ground-truth behaviour but at small scale.

Empirical studies report significant differences on identical topologies: 10–30 % over- or under-estimation in packet delivery ratio between simulators and testbeds [4], up to 20 % in energy consumption between emulators and testbed measurements before simulator calibration [5], and timing errors in Cooja/MSPSim from about 15 % on TelosB to nearly 2× on Zolertia Z1 [6]. These are structural, not outliers. The sound response is *cross-backend validation*: run the same experiment on two or more categories and reason about the differences, as community position papershave long argued [7], [8]. Yet such evaluations remain rare because the tooling makes them too costly.

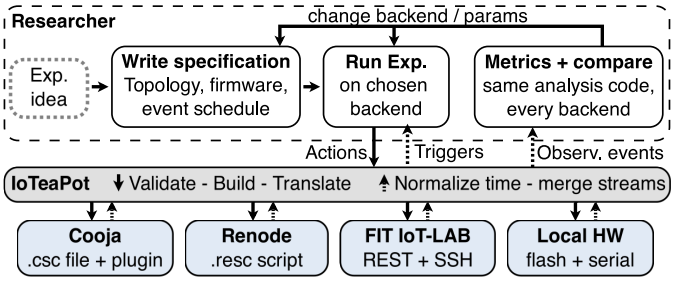


Fig. 1: Overview of IoTeaPot: one experiment, every backend, all observations on one timeline.

Every extra backend has a real cost: an *engineering* cost, since each backend describes topology, captures serial output, and schedules events in its own way, so the experiment must be re-encoded against a fresh set of incompatible interfaces; a *semantic* cost, since each reports observations in its own clock domain and granularity, which must be reconciled before any comparison is valid; and a *methodological* cost, since every re-implementation is a fresh risk to introduce silent errors that undermine the comparison.

This paper presents IoTeaPot, a framework that separates experiment specification from execution. It rests on a single thesis: *the experiment specification (topology, firmware, schedule, observation contract) does not depend on the chosen backend; only the translation layer changes*. The researcher writes the experiment once; the framework translates it multiple ways. Figure 1 shows what this looks like in use: from

a single experiment description and the metrics to compute, IoTeaPot validates the experiment, generates each target's native artefacts, runs it, and returns every observation on one normalised timeline.

The contribution is not a new simulator, emulator, testbed, or portal, but an *orchestration layer* above the artefacts existing tools already consume: Cooja, Renode, FIT IoT-LAB, and probe-based hardware stay intact, used as intended. These claims are scoped to the *orchestration* level (composition, scheduling, fault injection, observation); multi-hop RF realism, scale, and energy are out of scope (Section IV). The framework is available at https://github.com/jeremydub/ioteapot/.

Existing framework-level work falls into three categories, none in the niche we target. **Co-simulators and hybrid bridges** such as Whitefield [9] and IoTNetEMU [10] couple several execution mechanisms internally yet expose a single input format: each is one composite backend that cannot re-target one experiment across distinct backend *categories*. **Testbed federations** such as SLICES [11] present uniform reservation surfaces but do not unify experiment specification. **Generic experiment-control frameworks** such as OMF [12] and nepi-ng [13] are tied to specific testbed families or to broader networked systems rather than low-power WSNs; the closest, the POS controller [14], runs one specification unchanged across the Chameleon and CloudLab testbeds, but targets general bare-metal networking testbeds, not the simulator/emulator/testbed/hardware split of low-power WSNs. IoTeaPot is a specification-and-orchestration layer *above* simulators, emulators, testbeds, and hardware, replacing none of them, translating one description into whatever each tool already consumes.

II. A Generic Experimentation Model

A. Experiment Model

We decompose an experiment into five orthogonal, backend-agnostic abstractions sufficient for its static composition: **Node**, **Firmware**, **Application**, **Topology**, and **Radio Medium**.

A **Node** represents a single mote at a position (x, y, z). It carries either a pre-compiled **Firmware** (an opaque binary) or a source-level **Application** (compiled on demand by the right build system). A Node also declares how its output reaches the framework and whether it powers on at experiment start or waits for an explicit start event, enabling staged-boot and fault-injection scenarios.

A **Topology** groups Nodes with a **Radio Medium** whose reception rule, defined for every node pair, determines which nodes hear which (e.g. a unit-disk model with transmitting and interference ranges). An **Experiment** ties a Topology to an event schedule; it is the unit a researcher hands to any backend.

Listing 1 (lines 1–15) composes these abstractions into a complete experiment: two Contiki-NG applications built from source (a UDP server and client), each wrapped in a Node, placed in a topology under a unit-disk radio medium, and bound to an Experiment.

```
1  # Contiki-NG applications built from source
2  server_app = contiki.Application(
3      app_folder="path/to/udp-app", target="udp-server")
4  client_app = contiki.Application(
5      app_folder="path/to/udp-app", target="udp-client")
6
7  # Two nodes: server auto-starts, client held off
8  server = Node(identifier="server", position=(0, 0, 0),
9                application=server_app)
10 client = Node(identifier="client", position=(2, 0, 0),
11                application=client_app, auto_start=False)
12
13 # Building a topology of 2 nodes
14 topology = Topology(medium=UnitDiskRadioMedium(
15     transmitting_range=20, interference_range=0))
16 topology.add_node(server)
17 topology.add_node(client)
18
19 # Create a generic experiment from topology
20 exp = Experiment(topology=topology, name="validation")
21
22 # Time-driven: start client, then arm periodic traffic
23 exp.after(10).do(StartNodeAction(node=client))
24 exp.after(30).do(SerialWriteAction(
25     node=client, text="send-data"))
26
27 # Output-driven: stop server on its third reception
28 exp.when(SerialReadTrigger.contains(
29     server, "hello 3")).do(StopNodeAction(node=server))
30 exp.after(60).do(StopExperimentAction())
31 exp.run(Renode(), event_listener=PrintListener())
```

Listing 1: A complete experiment: static composition (lines 1–15), event schedule (lines 16–23), and execution (line 24).

TABLE I: Feature support across execution environments. ✓ = supported, — = not supported.

Capability	Cooja	Renode	IoT-LAB	Local
Topology placement	✓	✓	—	—
Radio medium config.	✓	✓	—	—
Serial output streaming	✓	✓	✓	✓
Serial input (write)	✓	✓	✓	✓
Radio packet capture	✓	✓	✓	✓
Node start/stop	✓	✓	✓	✓
Node mobility	✓	✓	—	—
RTT log transport	—	—	—	✓

B. Event and Observation Model

An experiment is also its dynamic behaviour, where the engineering and semantic porting costs of Section I concentrate.

a) Triggers and actions: An **Event** is a (trigger, action) pair. Two trigger types cover the time-driven and output-driven control patterns: a *time trigger* that fires at a given offset from experiment start, and a *serial-read trigger* that fires when a node's serial output matches a textual condition (contains, starts-with, or equals). Five action types are enough to express the orchestration patterns we have met: stop the experiment, start or stop a node mid-run (for fault injection), inject a command into a node's serial input, and relocate a node (for mobility studies). The vocabulary is deliberately small; depending on the backend, each action either maps directly onto a native primitive, is realised through a documented approximation, or is declared unsupported (Table I), rather than being silently emulated.

b) Observation surface: The observation surface is the narrow contract through which backends return output to the researcher. It exposes five events: experiment start, experiment end, a line of serial output from a node, a line written to a node, and a captured radio packet. Two implications follow.

First, timestamps need normalisation: each backend reports time from a different source (simulated clocks, emulator ticks, wall-clock on testbeds and hardware), so all observations are reconciled into a single unit and a common epoch before they reach the researcher.

Second, the narrow contract *separates measurement from experimentation*. An event listener is plain user code consuming these events to compute whatever the researcher cares about (PDR, end-to-end latency, convergence time), and the same experiment re-runs with a different listener without touching topology, firmware, or schedule.

c) Programming surface: Both are exposed through a fluent event-scheduling API (`after`/`when` in Listing 1) and an attachable event listener (plain Python classes whose methods receive the five observation events); lines 16–23 of Listing 1 exercise both trigger types and four of the five action types. The Experiment exposes two entry points: `run(environment)` for direct execution and `export(target, path)` for self-contained, framework-independent folders. Swapping `Renode()` for `Cooja()`, `Local()`, or `FitIotLab(...)` on the last line is the only change needed to re-target; the same listener, which covers the observation surface, is used for each backend.

III. Architecture

The framework is organised into four layers in a strict dependency hierarchy: each depends only on the layer(s) above it, so the experiment model (Layer 1) knows the minimum about any backend (Figure 2). Layer 1 realises the generic model of Section II as plain Python classes with no reference to any backend.

A. Layer 2: The Platform Catalog

The platform catalog is the framework's knowledge base about hardware, written as declarative configuration, not code. Each entry maps a sensor board to the attributes that build systems, backends, and device-management pipelines need: MCU family, per-OS build options, flash and control backends, USB vendor/product identifiers, and log transport modes.

Adding a board is a configuration change (the framework generates the matching platform class at import time), and the catalog is user-extensible, since an environment variable merges extra files with the built-in catalog, without touching the framework.

B. Layer 3: RTOS Integrations

The OS integration layer wraps the build systems of three embedded operating systems: Contiki-NG [15] and RIOT [16] (both GNU Make), and Embassy [17] (Cargo). Each wrapper produces a firmware binary from an Application object by invoking the RTOS's build system, caching it per (application, platform, configuration) triple.

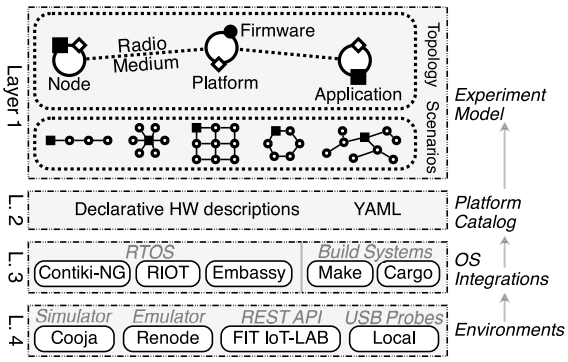


Fig. 2: The four-layer architecture; each layer depends only on the layer(s) above it.

Each Application also carries an OS-specific build configuration (a Contiki-NG `project-conf` object, Cargo features for Embassy) and build-time environment variables, applied before the build: a uniform handle on any RTOS knob reachable through the build environment, enabling parameter sweeps without editing project files.

C. Layer 4: Environments

An **Environment** models a backend that can execute an experiment. It is an *adapter*: it exposes a fixed set of *interfaces* the rest of the framework calls and implements each against one backend. Every environment honours the same contract (*validate* the model, then *execute* it by driving those interfaces), so adding a backend means implementing the interfaces, not touching any layer above.

Six interfaces, in two groups, recur across backends. *Provisioning* turns the static model into backend input: a **scenario descriptor** (topology, positions, radio medium) and **firmware deployment** (the binary onto each node's runnable). *Runtime* drives the dynamic surface on one timeline: the **log transport** (channel and codec), **radio capture** (packets routed to the uniform `radio_packet` callback), the **control plane** (triggers and the five actions), and the **time base** (native clock mapped to one epoch and unit). The interfaces are identical everywhere; only the machinery behind them differs (Table II).

A simulator or emulator *is* its own medium, clock, and log source, so for Cooja and Renode these interfaces are essentially the script the framework generates (a `.csc` or `.resc` carrying topology, medium, and schedule) plus the engine it launches, reads, and drives; physical backends have no such engine and must drive real machinery instead.

IV. Capabilities and Limits

The experiment of Listing 1 runs unchanged on each of the sixteen board/backend/OS combinations of Table III. Where a board recurs across backends, each runs the identical firmware image, so portability holds on one binary, not on re-implementations. Table I lists which modelled capabilities each backend supports.

TABLE II: The six interfaces each backend must drive, and the backend-specific machinery behind each.

Interface	**Cooja** (sim.)	**Renode** (emu.)	**FIT IoT-LAB** (testbed)	**Local** (hardware)
Scenario descriptor	`.csc` (UDGM grid+ranges)	`.resc` (machine per node)	REST reservation	auto USB discovery
Firmware deploy.	path in `.csc`	`LoadELF` in `.resc`	REST upload + flash	probe flash (catalog)
Log transport	Cooja engine TESTLOG	emulated UART	`serial_aggregator`+SSH	log reader (serial/RTT)
Radio capture	sim. medium	sim. medium	sniffer node → `.pcap`	nRF52840 USB sniffer
Control plane	IoTeaPot plugin	monitor cmds	remote script + REST	USB control drivers
Time base	μs → ms	ns → ms	wall-clock → rel.	host / device clock

TABLE III: Board/backend/OS combinations currently supported (C: Contiki-NG, R: RIOT, E: Embassy).

Board	**Cooja**	**Renode**	**IoT-LAB**	**Local**
CoojaMote	C	—	—	—
Zolertia Z1	C	—	—	—
CC2538-DK	—	C, R	—	—
Firefly	—	C, R	—	C, R
IoT-LAB M3	—	—	C, R	—
nRF52840-DK	—	—	C, R, E	C, R, E

By design, the uniform model cannot express every backend feature, and this boundary is not arbitrary. A capability is portably abstractable when its semantics live in the model's own vocabulary (nodes, firmware, events, packets, time) and not when they are fixed by the physical substrate (silicon timing, crystal behaviour, RF propagation, vendor peripherals). The framework turns this into an operational contract, reported for each capability at the pre-flight validation step (Section III-C): *exact* (the backend implements the model semantics directly), *approximate* (similar behaviour, with documented differences), *physical* (fixed by the deployment substrate, such as real RF or clock drift), *native-only* (outside the portable model, reached through the backend's own interface), or *unsupported* (the experiment is rejected, or a warning issued). Concretely, several things cannot be written in the portable model at all: per-node energy consumption (e.g. FIT IoT-LAB's power probes), MCU power states, arbitrary RF-propagation physics, and vendor-specific peripherals or debuggers; they are routed to a backend's native interface or rejected, rather than silently approximated.

Coverage of the unified surface is representative, not exhaustive: new boards, operating systems, backends, PHYs beyond 802.15.4, and richer radio media are added through catalog entries and build or capture wrappers, with no model change.

V. Conclusion and Outlook

This paper presented IoTeaPot, a framework that separates WSN experiment specification from execution backend to ease and promote cross-backend experimentation. A single backend-agnostic model captures the orchestration concerns common to simulators, emulators, public testbeds, and host-attached hardware: five static abstractions plus a narrow event-and-observation surface, realised by a layered architecture; one specification drives sixteen board/backend/OS combinations unchanged. The framework is best seen as a *methodology enabler*: it makes no backend faster or more faithful, but pays the porting cost of Section I *once*, not per backend pair. Cross-backend statistical comparison, uniform energy abstraction, and characterisation of the framework's own overhead are left as futur work; a quantitative study of the fidelity differences it exposes between backends is ongoing. Making such differences cheap to observe is what would let cross-backend validation move from a rare effort toward a routine check.

References

[1] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with COOJA," in *Proc. IEEE SenseApp*, 2006, pp. 641–648.

[2] Antmicro, "Renode: Functional simulation framework for embedded systems," https://renode.io, 2024, open-source emulation framework.

[3] C. Adjih, E. Baccelli *et al.*, "FIT IoT-LAB: A large scale open experimental IoT testbed," in *Proc. IEEE WF-IoT*, 2015, pp. 459–464.

[4] U. M. Colesanti, C. Crociani, and A. Vitaletti, "On the accuracy of OMNeT++ in the WSN domain: simulation vs. testbed," in *Proc. ACM PE-WASUN*, 2007, pp. 25–31.

[5] C. Haas, J. Wilke, and V. Stöhr, "Realistic simulation of energy consumption in wireless sensor networks," in *Proc. 9th European Conf. on Wireless Sensor Networks (EWSN)*, ser. LNCS, vol. 7158, 2012, pp. 82–97.

[6] K. Roussel, Y.-Q. Song, and O. Zendra, "Using Cooja for WSN simulations: Some new uses and limits," in *Proc. EWSN*, 2016, pp. 319–324.

[7] C. A. Boano, S. Duquennoy *et al.*, "IoTBench: Towards a benchmark for low-power wireless networking," in *Proc. IEEE CPSBench*, 2018.

[8] R. Jacob, C. A. Boano *et al.*, "Towards a methodology for experimental evaluation in low-power wireless networking," in *Proc. ACM Bench@CPSIoTWeek*, 2019, pp. 18–23.

[9] Whitefield Project, "Whitefield: NS-3/Contiki/RIOT/OpenThread/Zephyr WSN simulation environment," https://github.com/whitefield-framework/whitefield, 2018.

[10] J. Oliveira, F. Sousa, and L. Almeida, "IoTNetEMU – a framework to emulate and test IoT applications," in *Proc. 19th ACM Int. Symp. QoS and Security for Wireless and Mobile Networks (Q2SWinet)*. ACM, 2023, pp. 33–37.

[11] S. Fdida, N. Makris, T. Korakis *et al.*, "SLICES, a scientific instrument for the networking community," *Comput. Commun.*, vol. 193, pp. 189–203, 2022.

[12] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, "OMF: A control and management framework for networking testbeds," in *Proc. ACM ROADS*, 2009, pp. 54–59.

[13] T. Parmentelat, T. Turletti, W. Dabbous, M. N. Mahfoudi, and F. Bronzino, "nepi-ng: An efficient experiment control tool in R2lab," in *Proc. ACM WiNTECH*, 2018, pp. 56–65.

[14] H. Stubbe, S. Gallenmüller, and G. Carle, "A methodology for reproducible and portable experiment workflows," *Comput. Commun.*, vol. 239, p. 108178, 2025.

[15] G. Oikonomou, S. Duquennoy *et al.*, "The Contiki-NG open source operating system for next generation IoT devices," *SoftwareX*, vol. 18, p. 101089, 2022.

[16] E. Baccelli, C. Gundogan *et al.*, "RIOT: An open source operating system for low-end embedded devices in the IoT," *IEEE Internet Things J.*, vol. 5, no. 6, pp. 4428–4440, 2018.

[17] Embassy contributors, "Embassy: The next-generation framework for embedded applications in Rust," https://embassy.dev, 2024.