CupCarbon Klaines

Agentic Digital Twin IoT Designer

User Guide — Version 8.3

A Smart City and IoT Wireless Sensor Network simulator with agentic artificial intelligence: design your network on a real city map, program every node in Python, and let AI agents connected to real large language models drive your digital twin — live.

Prof. Ahcene Bounceur
University of Sharjah — bounceur@gmail.com

www.cupcarbon.com

2026 — Free software (GNU GPL)

How to cite CupCarbon. If you use CupCarbon in academic work, please cite:

K. Mehdi, M. Lounis, A. Bounceur and T. Kechadi, "CupCarbon: A Multi-Agent and Discrete Event Wireless Sensor Network Design and Simulation Tool", 7th International ICST Conference on Simulation Tools and Techniques (SIMUtools'14), Lisbon, Portugal, 2014.

CupCarbon® — © 2026 — distributed under the GNU General Public License.
This guide documents CupCarbon Klaines 8.3 and supersedes the U-One 5.1 and IoT Programming 7.2 guides.

Contents

I Getting Started **3**

1 Introduction **4**

1.1 What is new in Klaines 8.3 4

1.2 Structure of this guide 5

2 Installation **6**

2.1 Requirements 6

2.2 Choosing your package 6

2.3 Platform-specific steps 6

2.3.1 Windows 6

2.3.2 macOS (Apple Silicon and Intel) 6

2.3.3 Linux (x64 and ARM) 7

2.4 Where CupCarbon stores its data 7

3 First Project **8**

3.1 Create the project 8

3.2 Deploy IoT nodes 8

3.3 Write the first script 8

3.4 Run 8

II The CupCarbon Environment **9**

4 The Interface and the Map **10**

4.1 The main window 10

4.2 Map interactions 10

4.3 One environment per project: WSN or IoT 10

4.4 Map backgrounds 11

5 The Objects **12**

5.1 Sensor and IoT nodes 12

5.2 Markers and routes 12

5.3 Buildings and radio visibility 12

5.4 Mobiles and natural events 12

III Programming the Nodes **13**

6 Python and the cup_* Functions **14**
6.1 Principle . . . 14
6.2 The Program editor . . . 14
6.3 Function reference . . . 14
6.3.1 Display and appearance . . . 14
6.3.2 Reading information . . . 15
6.3.3 Mobility . . . 15
6.3.4 Communication . . . 16
6.3.5 Agentic AI bridge . . . 16
6.3.6 Control . . . 16
6.4 Complete example: intelligent surveillance . . . 16
6.5 Julia and Node.js . . . 17

7 SenScript **18**

IV Simulation **19**

8 WSN Discrete-Event Simulation **20**

9 IoT Simulation **21**

V Agentic AI Workflows **22**

10 The Workflow Editor **23**
10.1 Concepts . . . 23
10.2 Tour of the editor . . . 23

11 Node Reference **24**
11.1 Triggers (no input port) . . . 24
11.2 Data and logic . . . 24
11.3 AI, CupCarbon and outputs . . . 25

12 The Data Model and Expressions **27**
12.1 Items . . . 27
12.2 Expressions . . . 27
12.3 Errors . . . 28

13 The AI Agent **29**
13.1 Providers and keys . . . 29
13.2 Structured answers and memory . . . 29

13.3 A complete chatbot 29

14 The CupCarbon–AI Bridge **30**

VI Remote Control and Integration **31**

15 The Command Console **32**

16 The Web Tools **33**

17 Driving CupCarbon from Claude **34**

18 Integration with Other Platforms **35**

Appendices **37**

A cup_* Quick Reference **37**

B Troubleshooting **38**

C Citing CupCarbon and License **39**

Part I

Getting Started

CHAPTER 1

Introduction

CupCarbon is a Smart City and Internet of Things Wireless Sensor Network (SCI-WSN) simulator. Its objective is to design, visualize, debug and validate distributed algorithms for monitoring and environmental data collection, and to create environmental scenarios such as fires, gas and mobile objects (vehicles, UAVs), within educational and scientific projects. It helps to visually explain the basic concepts of sensor networks, and supports scientists in testing wireless topologies, protocols and applications.

Networks are designed on an ergonomic interface built on OpenStreetMap: sensors are deployed directly on the map of a real city. Each node is programmed individually — in **Python** through the `cup_*` API (the modern way, Chapter 6) or in **SenScript** (the historical script, Chapter 7). The simulation is based on a discrete-event engine which takes into account radio propagation (ZigBee, LoRa, WiFi), interference, energy consumption, mobility, and the urban environment (buildings).

1.1 What is new in Klaines 8.3

The Klaines generation is the most important evolution of CupCarbon since its creation. In one sentence: *the digital twin now thinks.*

- **Agentic AI workflows** (Part V): a visual, n8n-style workflow editor built into CupCarbon — triggers, logic, MQTT and AI nodes connected on a canvas, running continuously and independently from the simulation.
- **Real LLM agents**: the AI Agent node connects to Anthropic Claude or OpenAI with your API key, with conversation memory, a chat panel, and an offline rule-based fallback.
- **The CupCarbon–AI bridge**: node scripts talk to workflows with `cup_toagent()` / `cup_fromagent()`.
- **Python `cup_*` API**: 33 predefined functions injected automatically at simulation start, with a syntax-highlighted editor, per-function help and examples.
- **Building-aware radio visibility**, now a toggle, recomputed *live* while nodes are dragged, with ranges identical to the classical circular ones.
- **Web remote control**: web console, CupCarbon Studio and a full web workflow editor, connected over MQTT with secure per-session topics; integration with n8n, Home Assistant and any MQTT platform.
- **Quality**: antialiased and throttled map rendering, non-blocking consoles, per-user settings storage, and one package per platform (Windows, macOS Intel/Apple Silicon, Linux x64/ARM).

1.2 Structure of this guide

Part I gets CupCarbon installed and a first project running. Part II presents the environment: the interface, the map and the objects. Part III covers the programming of the nodes in Python (and SenScript). Part IV explains the two simulations. Part V is dedicated to the agentic AI workflows. Part VI covers remote control and integration. The appendices provide quick references and troubleshooting.

CHAPTER 2

Installation

2.1 Requirements

Two free tools are required on every platform:

- **Java 17 or newer** — adoptium.net (Temurin). Verify with `java -version`.
- **Python 3** — python.org. Only needed to run IoT node scripts. Verify with `python3 -V` (Windows: `python -V`).

2.2 Choosing your package

CupCarbon Klaines is distributed as one zip per platform. Each zip contains the application (`cupcarbon.jar`, JavaFX included), a double-clickable launcher and the `utils/` runtime resources.

Platform	Package	Launcher
Windows x64	`cupcarbon_windows.zip`	`cupcarbon_win.bat`
macOS Apple Silicon (M1–M4)	`cupcarbon_macos_m.zip`	`cupcarbon_macm.command`
macOS Intel	`cupcarbon_macos_x.zip`	`cupcarbon_macx.command`
Linux x64	`cupcarbon_linux_x.zip`	`cupcarbon_linux.sh`
Linux ARM	`cupcarbon_linux_m.zip`	`cupcarbon_linux.sh`

Which macOS package? Apple menu → About This Mac: "Chip Apple M..." → Apple Silicon package; "Processor Intel..." → Intel package. The wrong choice produces an *incompatible architecture* error at startup.

2.3 Platform-specific steps

2.3.1 Windows

Install Java and Python (tick *Add python.exe to PATH*), unzip the package, double-click `cupcarbon_win.bat`. If SmartScreen warns, click *More info* → *Run anyway*.

2.3.2 macOS (Apple Silicon and Intel)

Install Java (choose the build matching your chip) and Python, unzip, double-click the `.command` launcher. The first time, macOS Gatekeeper may refuse: *right-click* → Open → Open.

2.3.3 Linux (x64 and ARM)

```
sudo apt install openjdk-21-jre python3   # Ubuntu / Debian / Raspberry Pi OS
unzip cupcarbon_linux_x.zip && cd cupcarbon_linux_x
chmod +x cupcarbon_linux.sh               # first time only
./cupcarbon_linux.sh
```

Use `uname -m` to identify your processor: `x86_64` → Linux x64 package, `aarch64` → Linux ARM package.

2.4 Where CupCarbon stores its data

What	Location
Settings (recent projects, executor paths, AI API keys)	the `utils/` folder next to the jar when writable; otherwise macOS `~/Library/Application Support/CupCarbon`, Windows `%APPDATA%\CupCarbon`, Linux `~/.config/cupcarbon`
Projects	wherever you create them — a project is a plain folder

To update CupCarbon, replace the application folder — projects and settings are elsewhere and are preserved. To uninstall, delete the folder (and optionally the settings folder).

CHAPTER 3

First Project

3.1 Create the project

`Project` → `New project`: choose a folder and a name. A CupCarbon project is a plain folder containing the network, the scripts, the GPS routes, the agentic workflows and the results — it can be moved, zipped and shared.

3.2 Deploy IoT nodes

Press key `2` then click on the map to add sensor/IoT nodes (each keyboard digit corresponds to an object type; see Chapter 5). Select a node and press `j` to choose its icon from the gallery.

3.3 Write the first script

Open the **Program editor** from the toolbar and type:

```
from cup_functions import *

while True:
    cup_print('Hello ')
    cup_mark()
    cup_wait(1)
    cup_print('World!')
    cup_unmark()
    cup_wait(1)
```

Save the script and assign it to the node. The import line gives you auto-completion in any IDE; when the simulation starts, CupCarbon replaces it with the real implementations automatically (Chapter 6).

3.4 Run

Click **Run IoT Simulation**: the node blinks and prints its messages. If nothing happens, open the **Executor Path Configuration** window and set the Python command (`python3`, `python`, or a full path).

Part II

The CupCarbon Environment

CHAPTER 4

The Interface and the Map

4.1 The main window

The main window is organized around the **map** (OpenStreetMap), where the whole network is designed. Around it: the **menu bar** (Project, Edition, Add, Display, Selection, Solver, Simulation, Map, Personal, Help), the **toolbar** (project shortcuts, Program editor, Run/Stop IoT Simulation, Agentic AI workflows, Command console...), the **parameters panel** on the left (device, radio, simulation parameters), and the **console** area.

The simulated time is displayed on the top-left corner of the map: red during a simulation (with a red frame around the map), green when finished. Message counters (sent, received, ACK, lost) are shown next to it; `ALT+D` toggles this display.

4.2 Map interactions

Action	How
Move the map	drag with the left mouse button
Zoom	mouse wheel
Add an object	press its digit key (e.g. `2` for sensor nodes) then click
Select an object	click on it
Select by rectangle	drag with the *right* mouse button
Invert selection	`i`
Duplicate the selection	`c`
Change an IoT node icon	select it, press `j`
Radio / sensing radius	`;` `,` and `(` `)` on the selection
Delete	`Del` (or the Edition menu)
Undo / Redo	`Ctrl+Z` / `Ctrl+Y`

4.3 One environment per project: WSN or IoT

A project contains *either* classical sensor nodes (WSN environment, SenScript, discrete-event simulation) *or* IoT nodes (IoT environment, Python `cup_*` scripts), never both — adding a node of the other family is refused with a message. The first node added defines the type, shown at the **bottom-left corner of the map** ("WSN" or "IoT").

All the controls follow the environment automatically:

- every **Run** button (toolbar, left panel, Simulation menu) starts the simulation matching the project, and every **Stop** button stops the one actually running;

- the editor buttons open the **Program editor** (Python) in an IoT project and the **SenScript editor** in a WSN project;
- *Select All Sensors* also selects the IoT nodes, and the Network Information panel counts IoT nodes and selected IoT nodes.

4.4 Map backgrounds

The `Map` menu switches between the online light and dark maps and several *offline* backgrounds (grid, white, black, book...) which work without any internet connection. Online tiles are downloaded with short timeouts: an unreachable tile server never blocks CupCarbon.

CHAPTER 5

The Objects

5.1 Sensor and IoT nodes

The main object of CupCarbon. Each node owns: a position (GPS), one or several **radio modules** (802.15.4/ZigBee, WiFi, LoRa — with channel, network address, radio and sensing radii), a **battery**, an optional **script** (Python or SenScript), and an icon. Nodes detect the mobiles and natural events that enter their sensing area (`cup_dsensor()`).

5.2 Markers and routes

Markers are clickable waypoints used to draw **routes**. A sequence of markers can be converted into a GPS route file assigned to a node — the node then follows the route during the simulation (`cup_routego()`, `cup_routestop()`, `cup_speed()`...). Markers are also used to delimit the area of the buildings import.

5.3 Buildings and radio visibility

Place two markers delimiting a rectangular area and click **Load buildings**: the real buildings of the city are downloaded from OpenStreetMap (Overpass) and drawn on the map.

The **Visibility** toolbar button then toggles the propagation mode:

- *OFF* (default): radio ranges are the classical circles.
- *ON*: each range is recomputed through the urban environment — buildings cut their shadows out of the coverage. The outer boundary is a true circle with exactly the same radius as the classical range, at any latitude. While a node is *dragged*, its zone is recomputed continuously and follows the movement live.

> The buildings download can take up to a few minutes for a large area — the request runs in the background with a generous timeout. If the Overpass server is busy, a clear message is displayed; retry a minute later or reduce the area.

5.4 Mobiles and natural events

Vehicles and flying objects follow routes; gas clouds and fires are natural events that sensors detect. Weather can influence the energy models.

Part III

Programming the Nodes

CHAPTER 6

Python and the cup_* Functions

6.1 Principle

Each IoT node runs a *real external Python script* — one process per node — connected to CupCarbon through a simple protocol. In Klaines you never manipulate this protocol directly: you start your script with

```
from cup_functions import *
```

and use the `cup_*` functions. The import gives auto-completion while editing (the module `utils/py/cup_functions.py` ships with CupCarbon); when the simulation starts, CupCarbon *injects* the real implementations into a temporary copy of the script — flushing, protocol and answers are handled for you.

> **Compatibility.** The historical low-level style (`def cup(com): print(com, flush=True)` then `cup("mark")` and `input()`) documented in the previous IoT guide still works unchanged — the `cup_*` functions are a cleaner layer on the very same protocol.

6.2 The Program editor

The editor (toolbar) provides Python syntax highlighting, automatic indentation (`Tab` = 4 spaces, auto-indent after `:`), tabs highlighted in red with a one-click *Tabs→Spaces* fix (Python forbids mixing), the full list of `cup_*` functions (double-click to insert), and a **Help** button giving, for every function, its description and three examples. Saving a file without an extension adds `.py` automatically.

6.3 Function reference

6.3.1 Display and appearance

`cup_print('message')`
displays a message next to the node on the map.

`cup_mark()` / `cup_unmark()`
marks (highlights) / unmarks the node.

`cup_visible(True|False)`
shows or hides the node.

`cup_pic(n)`
switches the node icon to picture number `n`.

`cup_image(url[, shape])`
uses the picture of a URL (PNG, JPEG, GIF, BMP) as the node icon, e.g. `cup_image('https://randomuser`
It is downloaded in the background (the simulation does not wait) and cached; `'circle'` (default) gives a round avatar, `'square'` keeps the whole image; `cup_image()` restores the built-in icon.

`cup_alert('message')`
shows a popup alert window.

`cup_message('message')`
shows a message in the console.

6.3.2 Reading information

All read functions *return typed values directly* — no `input()`:

`cup_getid()` / `cup_getname()`
node identifier (int) / name (str).

`cup_getx()`, `cup_gety()`, `cup_getxy()`
position (longitude, latitude) as floats.

`cup_ismarked()`
`True` if the node is marked.

`cup_dsensor()`
`True` if the sensor detects an event (gas, fire, mobile) in its sensing area.

`cup_getmessage()` / `cup_getaction()`
last received message / last user action on the node.

6.3.3 Mobility

`cup_move(lon, lat)` / `cup_moveto(lon, lat, z=0)`
moves the node to a GPS position.

`cup_route(m1, m2)`
follows the route between two markers.

`cup_routego()` / `cup_routestop()` / `cup_routewait(s)`
resumes / stops / pauses the assigned GPS route.

`cup_speed(ratio)`
changes the route speed.

`cup_rudder_angle(angle)`
steering angle for vehicle-like mobility.

`cup_mapcenter(lon, lat)`
moves the map so that the point (longitude, latitude) is at the center of the view (zoom unchanged); e.g. a node can keep itself in view with `x, y = cup_getxy(); cup_mapcenter(x, y)`.

`cup_drawroutes()`
draws all the routes of the project on the map (same as the *Draw all routes* button).

6.3.4 Communication

`cup_send('message')`
radio broadcast to the neighbors (`ALL` or `'*'` may be given explicitly).

`cup_send('message', dest)`
radio send to node `dest`, delivered *only if* `dest` is inside the radio range of the sender – otherwise the message is lost, as on a real link.

`cup_fsend('message'[, dest])`
forced send: same as `cup_send` but ignores the radio range (out-of-band channel such as a backbone or the Internet); with no destination it reaches every node of the project.

`cup_read()`
blocking receive; returns the message (str).

6.3.5 Agentic AI bridge

`cup_toagent('message', chid)`
sends a message to every running workflow whose *CupCarbon Read* trigger has this `chid` — the *Channel ID*, a number identifying the communication channel between node scripts and workflows (Chapter 14).

`cup_fromagent(chid)`
blocking wait for a message sent back by a *CupCarbon Send* workflow node with this `chid`.

6.3.6 Control

`cup_wait(seconds)`
pauses the script (simulated time aware).

`cup_stopsimulation()`
stops the whole IoT simulation.

6.4 Complete example: intelligent surveillance

A detector node monitors its zone and alerts a mobile camera node, which travels to the event:

```
# --- Node 1: detector -----------------------------------
from cup_functions import *

while True:
    if cup_dsensor():          # gas / fire / mobile detected
        cup_mark()
        x, y = cup_getxy()
        cup_send(f'{x}#{y}', 2)    # alert node 2 (no spaces: '#')
        cup_wait(5)
        cup_unmark()
    cup_wait(0.5)
```

```
# --- Node 2: mobile camera ------------------------------
from cup_functions import *
```

```
while True:
    msg = cup_read()              # blocking
    x, y = map(float, msg.split('#'))
    cup_print('Moving to the event')
    cup_moveto(x, y)
    cup_pic(2)                    # switch to "camera on" icon
```

6.5 Julia and Node.js

Scripts can also be written in Julia (`.jl`) and JavaScript/Node (`.js`) using the same low-level protocol; examples ship in `utils/jl` and `utils/js`. The executor paths of the three languages are set in the Executor Path Configuration window.

CHAPTER 7

SenScript

SenScript is the historical, built-in script of CupCarbon sensor nodes — no external language required. Each line is a command (`set`, `send`, `read`, `mark`, `move`, `if`/`while` control structures, variables with `$`...). It remains fully supported: existing projects and the 21 examples of the previous guide run unchanged, and the WSN discrete-event simulation (Chapter 8) is driven by SenScript or Python equally.

SenScript scripts are written in the **SenScript editor** — the same window as the IoT Program editor (syntax highlighting, line numbers, command list with per-command help and examples, Blink example). Files are saved in the project's scripts folder with the `.csc` extension added automatically when missing, and only `.csc` files are listed. In a WSN project, every editor button opens this editor.

For the complete SenScript command reference, keep the previous guide (*CupCarbon User Guide U-One*) at hand — Klaines does not change the language. New projects targeting IoT scenarios are encouraged to use Python (Chapter 6).

Part IV

Simulation

CHAPTER 8

WSN Discrete-Event Simulation

The WSN simulation is a faithful discrete-event engine: every radio message, sensor event and timer is scheduled on the simulated clock. It takes into account the radio standard of each module (802.15.4/ZigBee, WiFi, LoRa), the propagation and visibility model (with buildings, Section 5.3), a statistical MAC layer, optional interference at the PHY layer (α-stable models), the clock drift, and the **energy consumption** of each node (configurable models; battery levels can be plotted against simulated time and saved in the project results).

Simulation parameters (duration, speed, arrows, logs...) are set in the left panel; the **Visibility** checkbox of the simulation parameters activates the building-aware propagation during the simulation itself.

CHAPTER 9

IoT Simulation

The IoT simulation (toolbar button **Run IoT Simulation**) runs each node's external script — one real process per node, in parallel. It is the mode used with the Python `cup_*` API, and the one that interacts with the agentic workflows through the bridge. **Stop IoT Simulation** terminates all node processes cleanly.

Both simulations are independent from the agentic workflows: workflows are started and stopped with their own Run/Stop buttons (Chapter 10) and keep running while simulations start and stop.

Part V

Agentic AI Workflows

CHAPTER 10

The Workflow Editor

10.1 Concepts

An *agentic workflow* is a graph of nodes connected on a canvas — exactly like n8n: **trigger** nodes start an execution, and the data then flows from node to node, each one transforming, routing or acting on it. Workflows are stored inside the project (`agents/` folder, one JSON file per workflow, compatible with the kagent web format) and run *continuously* until stopped — independently from the simulations.

10.2 Tour of the editor

Click the robot button in the toolbar. The editor opens maximized, in dark mode:

- **Workflows bar**: create, load, save, delete workflows; every workflow of the project is listed.
- ≡ **Nodes**: toggles the node palette — double-click a node type to add it at the center of the current view.
- **Canvas**: drag nodes to arrange them; drag from a node's output port to another node's input port to connect (forward connections are smooth curves, backward ones are routed orthogonally). Drag the empty canvas to pan; the middle button zooms (default zoom 100%).
- **Properties panel**: select a node to edit its configuration.
- **Input | Output panels** (bottom, as in n8n): the last item array *received* and *produced* by the selected node, side by side, each with its item count and a **Table** view (one row per item, one column per field) or a **JSON** view. They refresh live while the workflow runs and stay readable after it stops.
- **Run / Run All / Stop / Stop All**: start and stop workflows. Running workflows are highlighted in green and locked against edits.
- **Chat panel**: appears when the workflow contains a Chat Trigger; conversation bubbles, cleared at each run.
- **API Keys**: stores your Anthropic / OpenAI keys (Chapter 13).

CHAPTER 11

Node Reference

11.1 Triggers (no input port)

Manual Trigger
fires once when the workflow starts; initial items can be given as JSON.

Schedule
fires periodically (interval in seconds).

MCP Server Trigger
exposes the workflow as *one MCP tool* (name, description, input fields) on `http://localhost:5690/mcp` (Streamable HTTP; local only by default, optional bearer token). When Claude or any MCP client calls the tool, its arguments become the item, the workflow runs, and the items reaching the ends of the branches are returned as the tool result. In Claude Desktop: `npx mcp-remote http://localhost:5690/mcp` as the server command.

Chat Trigger
fires when you send a message in the chat panel; the item carries `{"chatInput": "..."}`.

MQTT Trigger
subscribes to a broker/topic and fires on every received message (`{"message", "topic"}`).

CupCarbon Read
fires when a node script calls `cup_toagent(msg, chid)` with the matching `chid` (Channel ID) (`{"message", "chid"}`). Several workflows may share the same `chid`.

11.2 Data and logic

Edit Fields (Set)
sets or overwrites fields on every item — structured rows (name, value, type), values may use expressions.

Code (Function)
JavaScript code, two modes: *once for all items* or *once per item* (Chapter 12).

API Call
calls a REST API for each item: base URL + path, query parameters (JSON), and an authentication preset — *bearer* token, API key in a *header* or in the *query*, or *basic* (`user:password`). Same response handling as HTTP Request. Keys are stored in the workflow file: do not share workflows that contain secrets.

HTTP Request
one request *per item* (GET, POST, PUT, PATCH, DELETE); URL, headers (JSON) and body accept `{{ }}` expressions evaluated on that item. The response (parsed as JSON when

possible) is stored in a field of the item (default `response`) together with `statusCode`; a status ≥ 400 is an error (or an `error` field with *Continue on fail*).

If
routes every item to the `true` or `false` output port according to a condition.

Filter
keeps only the items matching the condition.

Split Out
splits an array field into one item per element.

Aggregate
collapses all items into one (collecting a field into an array).

Merge
combines two branches: *append*, *by position* or *by key*, with an optional wait timeout.

Wait
pauses the flow for a delay.

11.3 AI, CupCarbon and outputs

AI Agent
the LLM node — Chapter 13.

MCP Client
calls a tool of an external MCP server for each item, over *http* (server URL, optional bearer token) or *stdio* (a local command such as `npx -y @modelcontextprotocol/server-filesystem /tmp`). Leave *Tool name* empty to list the server's tools. Arguments are JSON with `{{ }}` expressions, e.g. `{"message": "{{ $json.message }}"}`; the result is stored in the `mcp` field.

CupCarbon Send
answers the node scripts listening in `cup_fromagent(chid)`. Delivery is *real-time*: the message reaches only the nodes listening on this channel at that moment (a node becomes a listener at its first `cup_fromagent` call); with no listener the message is lost. Messages accepted by a listening node are kept in that node's own mailbox and read in order by its next `cup_fromagent` calls.

CupCarbon Action
executes a simulator action on a node. The properties panel offers one button per available function (`mark`, `unmark`, `print <text>`, `moveto <lon> <lat> [z]`, `visible 0|1`, `pic <n>`): a click fills the command with default parameters, freely editable (expressions allowed).

Sensor Input
reads live values from a simulator node (position, battery, detection...) into the item.

MQTT Publish
publishes the item (or a template) to a broker/topic; protocol (`ws`/`wss`/`tcp`/`ssl`), host and port are separate fields with automatic default ports.

Display
shows the items in the output viewer (debugging).

Chat Response
sends the answer back to the chat panel.

CHAPTER 12

The Data Model and Expressions

12.1 Items

Exactly as in n8n, the data flowing between nodes is an **array of items**; every item is a JSON object. A node receives the items of the previous node, processes them — by default *one by one*, preserving order — and outputs a new array. An empty array is valid and simply produces no work. Exceptions: *If* routes items, *Filter* drops items, *Split Out* multiplies them, *Aggregate* collapses them, and the *Code* node in all-items mode sees the whole array at once.

In other words, a node whose function is f turns

$$[\,\{a\},\{b\},\{c\}\,] \longrightarrow [\,f(\{a\}),f(\{b\}),f(\{c\})\,]$$

For instance, an *Edit Fields* node setting the *boolean* field `alert = {{ $json.t > 30 }}` on three sensor readings gives:

```
in : [ {"t": 25}, {"t": 33}, {"t": 41} ]
out: [ {"t": 25, "alert": false},
       {"t": 33, "alert": true },
       {"t": 41, "alert": true } ]
```

and an *HTTP Request* on those three items sends three requests. The Input and Output panels of the editor show exactly these two arrays.

12.2 Expressions

Configuration fields accept JavaScript expressions between double braces, evaluated for each item in a fully sandboxed engine (no file, network or system access):

```
{{ $json.temperature > 30 ? 'hot' : 'ok' }}
Sensor {{ $json.chid }} says {{ $json.message }}
```

- `$json` — the current item;
- `$itemIndex` — its index;
- `$input.all()` — all input items.

In the **Code** node, return the new items array (all-items mode) or the new item (per-item mode):

```
// mode: once per item
return { ...$json, name: $json.name.toUpperCase() };
```

12.3 Errors

A failing item stops the workflow with a clear message including the item index — unless *continue on fail* is enabled on the node, in which case the item is skipped and the flow continues.

CHAPTER 13

The AI Agent

13.1 Providers and keys

The AI Agent node sends each item (plus your prompt and the conversation memory) to a real LLM:

- **Anthropic Claude** or **OpenAI** — select the provider; the **Model** list then shows only that provider's models (fastest first): `claude-haiku-4-5-20251001`, `claude-sonnet-5-5`, `claude-opus-5-5`, `claude-fable-5-1` for Anthropic; `gpt-6-luna`, `gpt-6.1-sol`, `gpt-6-astra` for OpenAI. Another model id of the same provider can be typed in;
- **Reasoning effort** (`low` by default) applies to the thinking models (Claude 5.x, GPT-5 and later); for them CupCarbon sends no temperature (rejected by these models) and reserves at least 4096 tokens for reasoning plus answer. **Temperature** is used by the older models only;
- the API key comes from the **API Keys** dialog (stored locally in `aiagent.cfg` in your settings folder, never shipped) or the environment variables `ANTHROPIC_API_KEY` / `OPENAI_API_KEY` (which take priority);
- without any key (or offline), the node falls back to its **configurable rules** (substring → response pairs).

13.2 Structured answers and memory

The agent is instructed to answer with a single JSON object such as `{"action": "mark", "response": "Node 3 marked"}`; the fields are merged into the item, so the next nodes (If, CupCarbon Action...) can act on `$json.action`. A window-buffer **memory** keeps the recent conversation per workflow and node, enabling real chatbots with follow-up questions.

13.3 A complete chatbot

Chat Trigger → AI Agent → Chat Response: run the workflow and talk to your network in the chat panel. Add CupCarbon Action after the agent, and the LLM's `action` field drives the map.

CHAPTER 14

The CupCarbon–AI Bridge

The two `cup_*` functions of Section 6 connect the simulation to the workflows:

```
from cup_functions import *

while True:
    cup_toagent('temperature 42', 32)    # fires CupCarbon Read (chid 32)
    answer = cup_fromagent(32)           # waits for CupCarbon Send (chid 32)
    cup_print(answer)
    cup_wait(2)
```

Workflow side: **CupCarbon Read (chid 32)** → **AI Agent** → **CupCarbon Send (chid 32)**. Run the workflow, then run the IoT simulation: each measurement travels to the LLM and the decision comes back to the node — live. Through **MQTT Trigger** and **MQTT Publish**, the same loop extends to the outside world: n8n workflows, Home Assistant automations, ChatGPT/Gemini/Hermes-based agents, or any MQTT-speaking platform.

Part VI

Remote Control and Integration

CHAPTER 15

The Command Console

Click **Command** in the toolbar. The console executes simulator commands locally (e.g. `add iot 25.3020 55.4855`, `mark 1`, `run`...) and connects CupCarbon to an MQTT broker for remote control.

> **Session topics.** At every launch, CupCarbon generates a random topic base such as `cupcarbon/session3fa29c.../console`. Click **Copy Topic** and paste it into the web tools. Since the topic is unguessable, nobody else on the public broker can drive your simulator. After restarting CupCarbon, copy the new topic and reconnect.

The console shows the current session in the **Session** field: edit it and click **Apply** to use a custom session (letters, digits, `-`, `_`) — convenient for labs, at the price of the random-topic secrecy. The `gettopic` command returns the base topic, and the console answers *discovery requests* on `cupcarbon/discovery/request` so local tools (the Claude MCP server) can find the topic automatically.

Recent console commands: `list routes` (GPS routes of the project), `list devices` (now includes the IoT nodes), `delete markers` (all the markers in one shot), `set pic <id> <n>` (icon of a node, like the `j` key), `route draw` / `route hide` (draw or hide all the routes, like `cup_drawroutes()`).

CHAPTER 16

The Web Tools

Four pages (on the website, also shipped in `utils/web/`) connect to the desktop over MQTT — paste the copied topic and click Connect:

Web Console
all simulator commands from the browser, with responses.

Command Reference
every console command with three examples.

CupCarbon Studio
console + workflow viewer: *Get* downloads a workflow from CupCarbon (JSON + visual preview). Studio is *read-only* for workflows; the *Edit in the workflow editor* button opens the web editor with the same connection prefilled.

Web Workflow Editor
the full visual editor in the browser — same nodes, same JSON format — with Get/Send to synchronize with the desktop.

CHAPTER 17

Driving CupCarbon from Claude

All the Claude integrations are downloadable from cupcarbon.com/claude:

Claude Skills
two installable skills teach Claude to write correct node programs: `cupcarbon-coding.skill` (Python IoT, the 30 `cup_*` functions and patterns) and `cupcarbon-senscript.skill` (SenScript WSN). Install once (drag the file into a Claude chat and Save skill, or Claude → Settings → Skills); Claude Code users can instead copy the unzipped folders into `~/.claude/skills/`. Both also ship with CupCarbon in the `claude_skill/` folder.

MCP server
a Python server (`cupcarbon_mcp.py`) exposing the simulator to Claude Desktop / Claude Code as 42 tools: add nodes and whole topologies, move, delete, list, icons, routes, scripts, run/stop, workflows... It talks to the command console over MQTT; *"Connect to CupCarbon"* works without pasting anything thanks to the discovery mechanism. Full setup in the dedicated PDF guide (same page). Batch tools (`add_nodes`, `set_properties`, `move_nodes`, `delete_objects`, `mark_nodes`, `run_batch`) do many operations in one call, through the console multi-commands `multiadd`, `multiset`, `multimove`, `multidelete`, `multimark` and `batch`.

Combined, one sentence in Claude — *"create a ring of 6 IoT nodes, write the blink script, assign it and run"* — goes from idea to a living simulation.

CHAPTER 18

Integration with Other Platforms

Everything in CupCarbon speaks **MQTT**: the console protocol, the workflow MQTT nodes, and the node scripts (via `paho-mqtt` in Python if desired — `pip install paho-mqtt`). This makes the integration with the rest of your infrastructure natural: **n8n** (MQTT nodes), **Home Assistant** (MQTT integration and automations), **ESP32** and real devices, cloud AI agents (ChatGPT, Gemini, Hermes...). A CupCarbon simulation can therefore stand as the *digital twin* inside a real deployment: real sensors publish, the twin simulates and decides, actuators subscribe.

Appendices

APPENDIX A

cup_* Quick Reference

Function	Role
`cup_print('m')`	display a message next to the node
`cup_alert('m')` / `cup_message('m')`	popup alert / console message
`cup_mark()` / `cup_unmark()`	mark / unmark the node
`cup_visible(b)`	show or hide the node
`cup_pic(n)`	change the node icon
`cup_getid()` / `cup_getname()`	identifier / name
`cup_getx()` / `cup_gety()` / `cup_getxy()`	position
`cup_ismarked()`	marked state
`cup_dsensor()`	event detection in the sensing area
`cup_getmessage()` / `cup_getaction()`	last message / last action
`cup_send('m'[, dest])`	radio broadcast / send (range checked)
`cup_fsend('m'[, dest])`	forced send, ignores the radio range
`cup_read()`	blocking receive
`cup_move(lon,lat)` / `cup_moveto(lon,lat,z)`	move the node
`cup_route(m1,m2)`	follow the route between two markers
`cup_routego()` / `cup_routestop()` / `cup_routewait(s)`	route control
`cup_drawroutes()`	draw all the routes on the map
`cup_speed(r)` / `cup_rudder_angle(a)`	speed / steering
`cup_toagent('m', chid)`	send to the agentic workflows
`cup_fromagent(chid)`	blocking wait for a workflow answer
`cup_wait(s)`	pause
`cup_stopsimulation()`	stop the IoT simulation

APPENDIX B

Troubleshooting

Problem	Solution
CupCarbon does not start	check `java -version` (17+ required)
"incompatible architecture" error	wrong package for your processor; download the other macOS/Linux package and delete `~/.openjfx/cache`
macOS refuses to open the launcher	right-click → Open → Open (first time only)
Windows SmartScreen warning	More info → Run anyway
IoT scripts do not run	set the Python command in Executor Path Configuration; assign a script to each node
Python `TabError`	open the script in the editor and click Tabs→Spaces
Map empty / slow offline	choose an offline background in the Map menu
Buildings do not load	Overpass may be busy: retry, or reduce the area between the two markers
AI Agent answers "no rule matched"	the LLM was not reached: check the provider selected on the node and the API key
Web tool does not react	re-copy the session topic (it changes at every CupCarbon launch) and reconnect

APPENDIX C

Citing CupCarbon and License

CupCarbon is free software distributed under the GNU General Public License. If you use it in academic work, please cite:

```
@inproceedings{cupcarbon2014,
  title     = {CupCarbon: A Multi-Agent and Discrete Event Wireless
               Sensor Network Design and Simulation Tool},
  author    = {Mehdi, Kamal and Lounis, Massinissa and
               Bounceur, Ahcene and Kechadi, Tahar},
  booktitle = {7th International ICST Conference on Simulation
               Tools and Techniques (SIMUtools'14)},
  address   = {Lisbon, Portugal},
  year      = {2014}
}
```

See also: A. Bounceur et al., "CupCarbon-Lab: An IoT Emulator", IEEE CCNC, Las Vegas, USA, 2018. Website: cupcarbon.com.