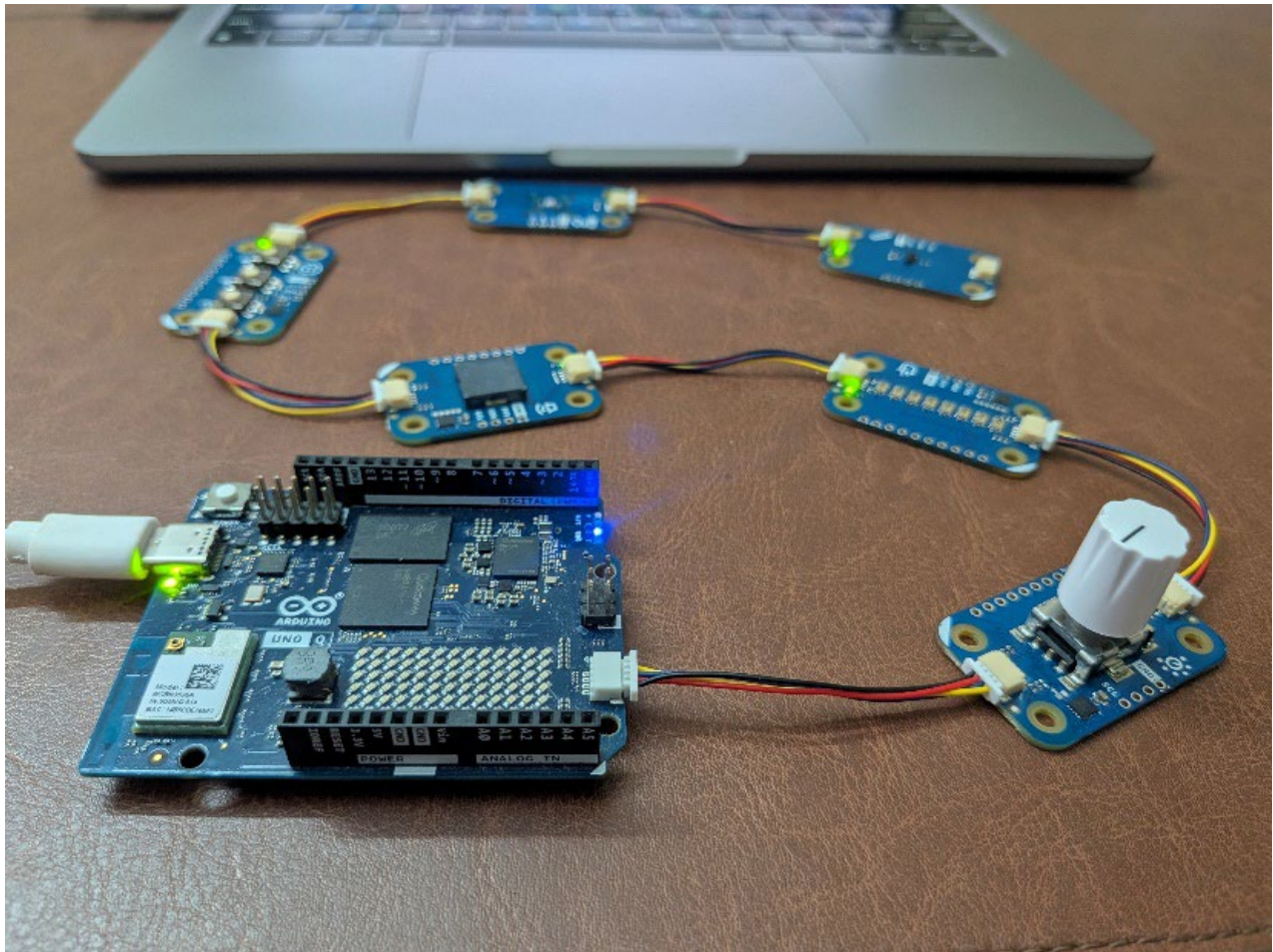


BUILD AN ON-DEVICE AI AGENT ON ARDUINO UNO Q

Natural-language hardware control using Arduino UNO Q, Arduino App Lab and a local LLM

Practical project guide | HatchAgent reference implementation



Arduino UNO Q connected to the Modulino sensor and actuator chain

WHAT YOU WILL BUILD

A fully offline AI agent that interprets natural-language commands, reads connected sensors and controls pixels and sound without cloud services or API keys.

How to use this guide

This guide is for embedded engineers, makers and application developers building a local conversational hardware interface. Follow the parts in order because the MCU firmware, Linux application, Web UI and model server are separate components that must work together.

IMPORTANT

The workflow modifies project configuration files and a locally installed library. Keep a backup of the original files and re-check the distance-sensor change after library updates.

Document map

Part	Purpose	Outcome
1. Plan	Architecture and requirements	Understand the two-domain system
2. Assemble	Connect UNO Q and Modulinos	Hardware chain ready
3. Build MCU side	Firmware, libraries and capabilities	Remote hardware functions exposed
4. Build MPU side	Web UI, assets and agent logic	Local application ready
5. Install AI	llama.cpp and model server	Local inference endpoint running
6. Validate	Run, test and troubleshoot	End-to-end agent verified
7. Extend	Fine-tuning and next steps	A path to custom behaviour

Contents

1. Project overview
2. Architecture and data flow
3. Requirements and readiness checks
4. Hardware setup
5. Create the App Lab project
6. Configure the MCU firmware
7. Apply the Distance Modulino fix
8. Enable the Web UI and copy assets
9. Add the agent logic
10. Install and start the local AI model
11. Run and validate the agent
12. Troubleshooting
13. Fine-tuning and extensions
14. Glossary and references

1. Project overview

1.1 Objective

Build HatchAgent, an on-device AI agent on Arduino UNO Q that translates natural-language instructions into hardware actions and sensor-driven rules. The application operates locally and combines deterministic microcontroller control with Linux-hosted application and language-model workloads.

1.2 Why an on-device agent?

Moving language-model inference to the edge allows the application to work without a continuous cloud connection or external API key. In this project, users can set pixel effects, trigger sounds, query sensors and create conditional automations using plain-language instructions.

1.3 Expected result

- A browser-based chat interface is available on the board.
- The local model interprets supported commands and returns structured actions.
- The MCU controls Modulino Pixels and Buzzer outputs.
- The application reads Buttons, Knob, Distance, Movement and Thermo inputs.
- Rules can connect sensor conditions to output actions without recompiling the firmware.

SUCCESS CRITERION

The system is ready when the interface reports “on-device, ready”, direct commands control the connected hardware, and sensor-based rules trigger the expected output.

2. Architecture and data flow

Arduino UNO Q separates real-time hardware work from application-level AI processing while the Bridge connects the two execution domains.

Domain	Responsibilities
MCU domain	Sensor acquisition; deterministic control loops; hardware interfaces; pixel effects; buzzer control; low-latency events
MPU / Linux domain	Python application; Web UI; local model integration; rule engine; data processing and orchestration
Bridge	Remote calls between Python on Linux and functions exposed by the MCU firmware
Browser	Natural-language input, status and interaction interface

DATA FLOW

User instruction → Web UI → Python agent → local model or rule engine → Bridge remote call → MCU → Modulino output.
Sensor values travel back through the Bridge to the rule engine and interface.

2.1 Software components

- sketch/sketch.ino: MCU firmware and remote hardware functions.
- sketch/sketch.yaml: MCU libraries and platform configuration.
- python/main.py: Web serving, model interaction, sensor monitoring and rule execution.
- app.yaml: application manifest, Web UI Brick and port.
- assets/: HTML, JavaScript, CSS and supporting browser libraries.
- llama.cpp model server: separate Linux process serving the local model on port 8080.

Reference implementation: [HatchAgent GitHub repository](#)

3. Requirements and readiness checks

3.1 Hardware

Items	MPN	Role
Arduino UNO Q 4 GB	ABX00173	Runs the MCU firmware, Linux application and local model
Modulino Buttons	ABX00110	Latched, debounced button input
Modulino Pixels	ABX00109	Solid, blink, pulse and off effects
Modulino Buzzer	ABX00108	Tone and melody output
Modulino Knob	ABX00107	Rotary input
Modulino Distance	ABX00102	Time-of-flight distance input
Modulino Movement	ABX00101	Movement input
Modulino Thermo	ABX00103	Temperature input
Qwiic cables	Supplied along with Modulino boards	Daisy-chain connections between modules

3.2 Software

- Arduino App Lab
- Arduino Bridge framework
- Arduino Modulino libraries
- Python runtime in the App Lab MPU environment
- llama.cpp
- Qwen2.5 1.5B GGUF model or the fine-tuned HatchAgent model
- Git and terminal access on the board

Project archive: [Download the HatchAgent master ZIP](#)

3.3 Pre-flight checklist

- ☐ UNO Q is configured and visible in Arduino App Lab.
- ☐ All seven Modulino boards and Qwiic cables are available.
- ☐ The board has network access for the initial repository, tool and model downloads.
- ☐ The Linux password is available for commands that request elevated permissions.
- ☐ The HatchAgent repository or ZIP is accessible.

4. Hardware setup

Connect the Modulino boards to the Arduino UNO Q using Qwiic cables. The source project uses Buttons, Pixels, Buzzer, Knob, Distance, Movement and Thermo modules in the chain.

1. Power off or disconnect the board before changing the Qwiic chain.
2. Connect each Modulino with the keyed Qwiic cables.
3. Check that every connector is fully seated and not offset.
4. Place modules so the cables are not strained.
5. Reconnect and power the UNO Q.

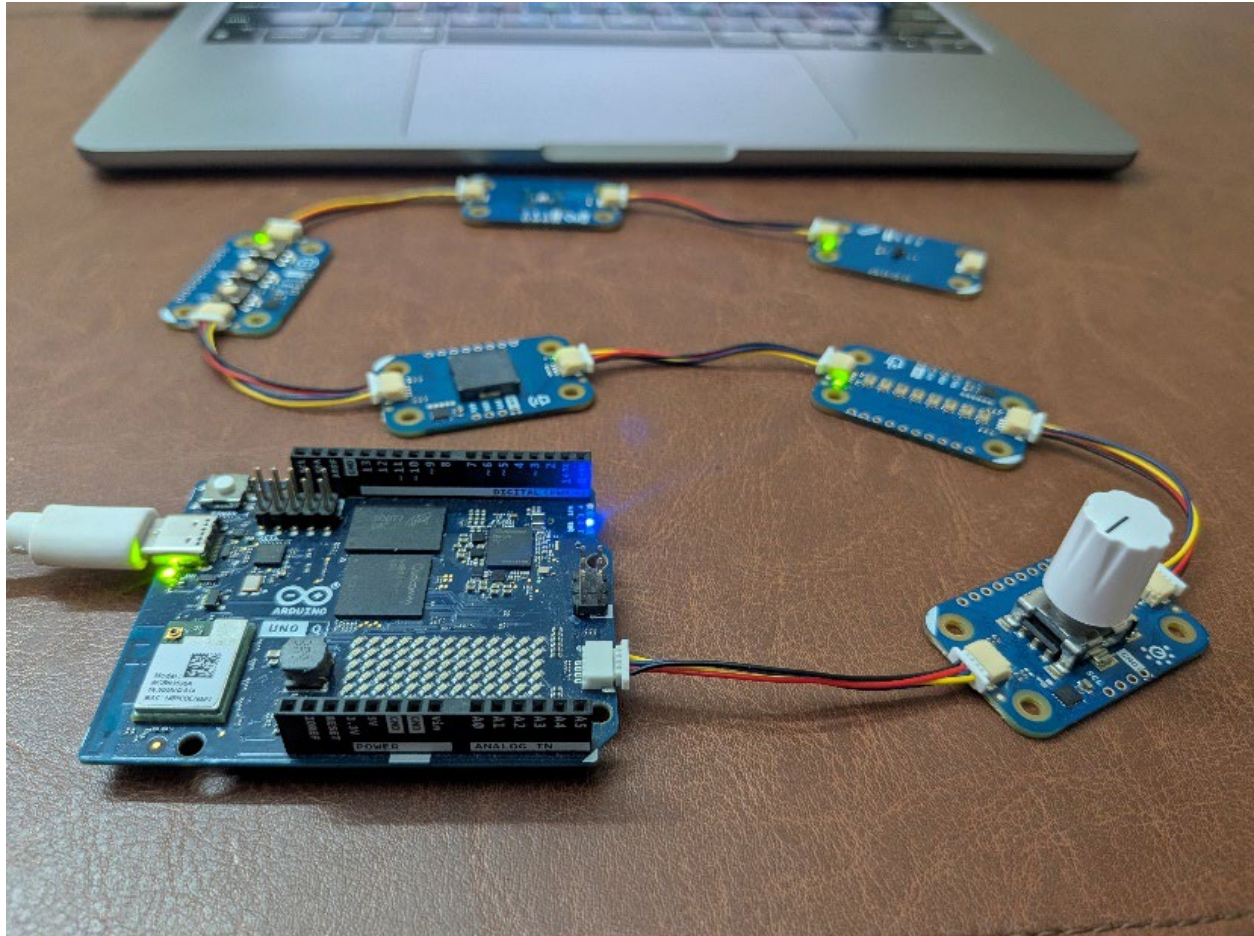


Figure 1. Example headless UNO Q and Modulino hardware setup

HARDWARE CHECK

If the final capability bitmask is lower than 127, reseal the Qwiic cables and rerun the application.

5. Create the App Lab project

1. Open Arduino App Lab and connect the UNO Q.
2. Open My Apps and select New App.
3. Name the project HatchAgent-test and choose the hatchling icon used by the source example.
4. Confirm that App Lab creates sketch/sketch.ino, sketch/sketch.yaml, python/main.py and app.yaml.
5. Locate the Bricks, Sketch Libraries and Files panels.

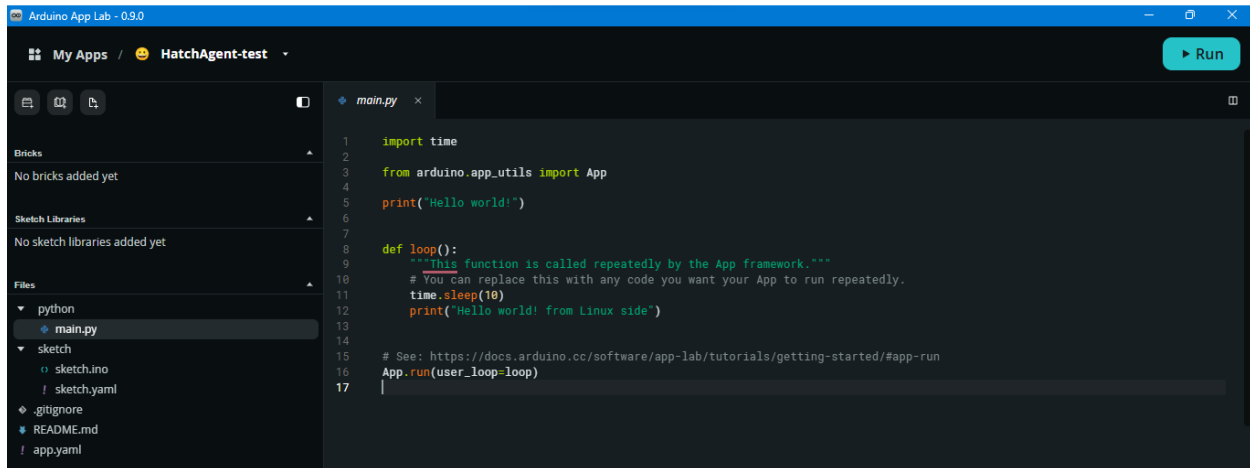


Figure 2. Blank HatchAgent-test application in Arduino App Lab

NAMING CONSISTENCY

The terminal commands in this guide use the folder hatchagent-test. If you choose another project name, replace the folder name consistently in every path.

6. Configure the MCU firmware

6.1 Add sketch.ino

1. Open sketch/sketch.ino.
2. Copy the full sketch.ino from the repository Code section.
3. Replace the blank sketch content with the supplied firmware.
4. Review the file and save it.

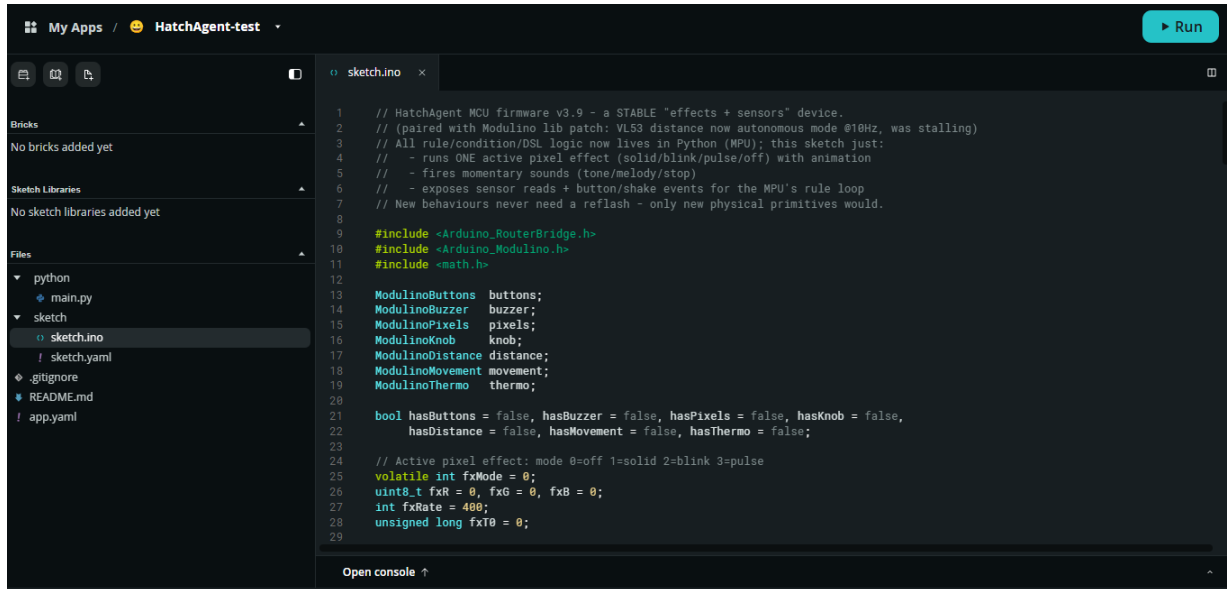


Figure 3. HatchAgent MCU firmware open in App Lab

6.2 Understand the exposed functions

The firmware reads each sensor once per loop, caches the values, latches and debounces buttons, controls one pixel effect at a time, and exposes hardware operations to the MPU as remote calls.

Remote function	Purpose
<code>set_effect()</code>	Control pixel output effects
<code>tone()</code>	Play a buzzer tone
<code>melody()</code>	Play a buzzer melody
<code>get_knob()</code>	Return the knob value
<code>get_distance()</code>	Return the distance value
<code>get_temp()</code>	Return the temperature value
<code>get_buttons()</code>	Return latched button state
<code>caps()</code>	Return the connected-capability bitmask

6.3 Update sketch.yaml

The source workflow edits sketch.yaml from the terminal because it is read-only in the App Lab editor.

```
nano ~/ArduinoApps/hatchagent-test/sketch/sketch.yaml
```

```

GNU nano 8.4 /home/arduino/ArduinoApps/hatchagent-test/sketch/sketch.yaml *
profiles:
  default:
    platforms:
      - platform: arduino:zephyr (0.55.2)
    libraries:
      - Arduino_RouterBridge (0.4.3)
      - Arduino_RPClite (0.3.0)
      - MsgPack (0.4.2)
      - DebugLog (0.8.4)
      - ArxTypeTraits (0.3.2)
      - ArxContainer (0.7.0)
      - Arduino_Modulino (0.8.0)
      - STM32duino VL53L4CD (1.0.5)
      - STM32duino VL53L4ED (1.0.1)
      - Arduino_LSM6DSOX (1.1.2)
      - Arduino_LPS22HB (1.0.2)
      - Arduino_HS300x (1.0.0)
      - Arduino_LTR381RGB (1.0.1)
  default_profile: default
  
```

Figure 4. Modulino libraries listed in sketch.yaml

BUILD CONFIRMATION

When the application runs, check the build log for confirmation that Arduino_Modulino and the required project libraries were added.

7. Apply the Distance Modulino fix

In the source project, an inter-measurement period of 0 can cause the Distance Modulino to return one reading and then stop updating. The documented workaround changes the period to 100 ms.

1. Open the App Lab terminal.
2. Open the Modulino.h file used by the installed Arduino_Modulino library.
3. Find `api->setRangeTiming(20, 0);` near the documented location.
4. Change it to `api->setRangeTiming(20, 100);`
5. Save the file and rerun the application.
6. Move a hand in front of the sensor and confirm that the value continues to change.

```
nano
~/arduino15/internal/Arduino_Modulino_0.8.0_cf7e8106889e770a/Arduin
o_Modulino/src/Modulino.h

api->setRangeTiming(20, 100); // 100 ms inter-measurement period
```

```
GNU nano 8.4 /home/arduino15/internal/Arduino_Modulino_0.8.0_cf7e8106889e770a/Arduino_Modulino/src/Modulino.h
delete tof_sensor;
tof_sensor = nullptr;
tof_sensor_alt = new VL53L4ED((TwoWire*)getWire(), -1);
ret = tof_sensor_alt->InitSensor();
if (ret == VL53L4ED_ERROR_NONE) {
    api = new _distance_api(tof_sensor_alt);
} else {
    delete tof_sensor_alt;
    tof_sensor_alt = nullptr;
    if (hubPort != nullptr) {
        hubPort->clear();
    }
    return false;
}
} else {
    api = new _distance_api(tof_sensor);
}

__increaseI2CPriority();
api->setRangeTiming(20,100);
api->startRanging();
if (hubPort != nullptr) {
    hubPort->clear();
}
return true;
}
operator bool() {
    return (api != nullptr);
}
bool available() {
    if (api == nullptr) {
        return false;
    }

    if (hubPort != nullptr) {
        hubPort->select();
    }

    uint8_t NewDataReady = 0;
    api->checkForDataReady(&NewDataReady);
```

Figure 5. Distance sensor timing change in Modulino.h

VERSION-SPECIFIC PATH

The folder name contains a library version and build identifier. If the exact path is different, locate the `Arduino_Modulino` folder actually referenced by the App Lab build rather than creating a new path.

8. Enable the Web UI and copy assets

8.1 Add the WebUI Brick

1. Open the Bricks panel.
2. Add the `arduino:web_ui / WebUI - HTML` Brick.
3. Review the Brick overview and confirm it appears in the project.

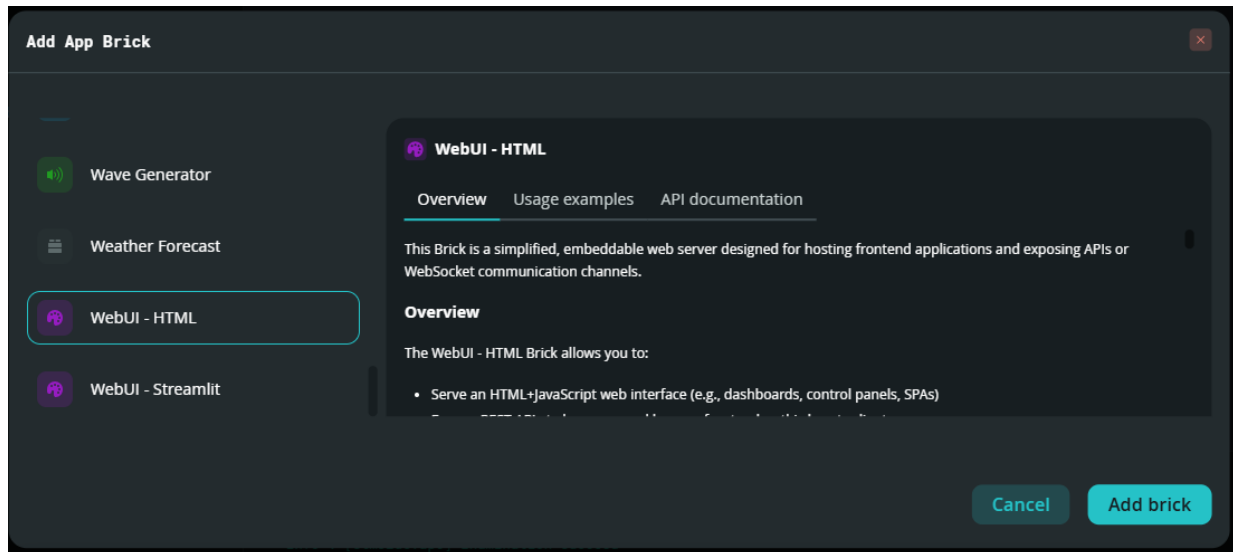


Figure 6. Adding the WebUI - HTML Brick

The source project uses this Brick to serve the HTML and JavaScript interface, expose application endpoints and support browser communication.

8.2 Configure app.yaml

Edit `app.yaml` in the terminal and set the application port. The example uses port 7000.

`nano ~/ArduinoApps/hatchagent-test/app.yaml`

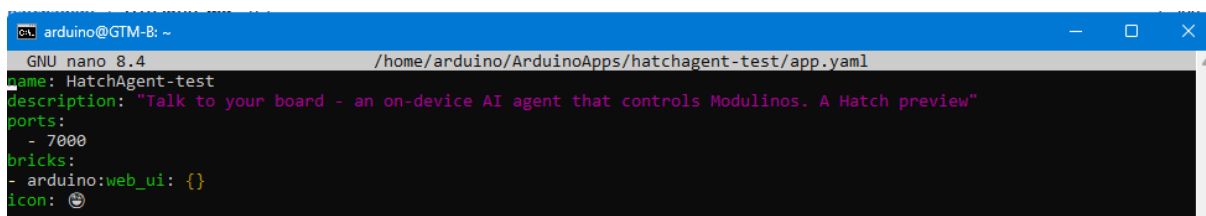


Figure 7. HatchAgent app.yaml with Web UI configuration

BROWSER ADDRESS

After the app starts, use the network URL printed in the Python log. The address follows the format `http://<device-ip>:7000`.

8.3 Copy the web assets

Clone the repository into a separate source folder, copy the contents of assets into the App Lab project, and verify the files and browser libraries.

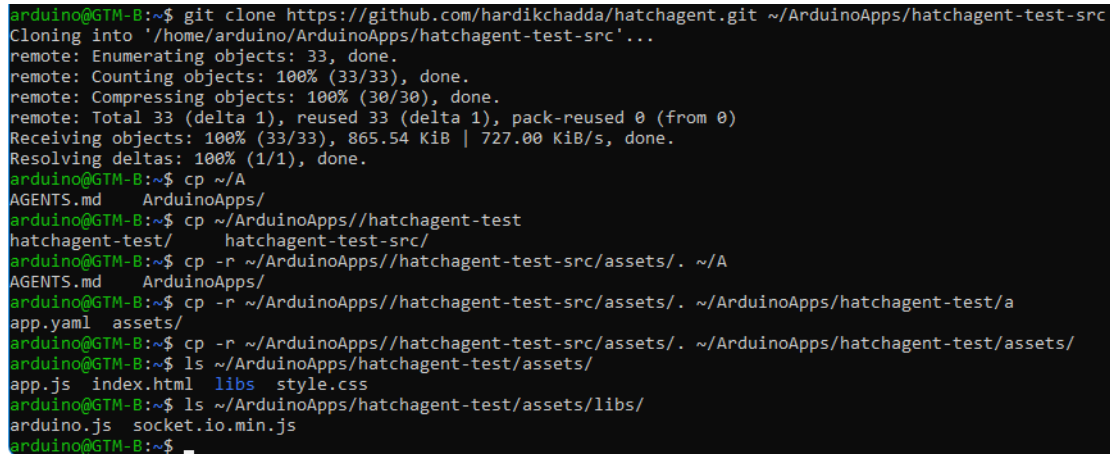
```
git clone https://github.com/hardikchadda/hatchagent.git
~/ArduinoApps/hatchagent-test-src
```

```
mkdir -p ~/ArduinoApps/hatchagent-test/assets
```

```
cp -r ~/ArduinoApps/hatchagent-test-src/assets/.
~/ArduinoApps/hatchagent-test/assets/
```

```
ls ~/ArduinoApps/hatchagent-test/assets/
```

```
ls ~/ArduinoApps/hatchagent-test/assets/libs/
```



```
arduino@GTM-B:~$ git clone https://github.com/hardikchadda/hatchagent.git ~/ArduinoApps/hatchagent-test-src
Cloning into '/home/arduino/ArduinoApps/hatchagent-test-src'...
remote: Enumerating objects: 33, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (30/30), done.
remote: Total 33 (delta 1), reused 33 (delta 1), pack-reused 0 (from 0)
Receiving objects: 100% (33/33), 865.54 KiB | 727.00 KiB/s, done.
Resolving deltas: 100% (1/1), done.
arduino@GTM-B:~$ cp ~/A
AGENTS.md  ArduinoApps/
arduino@GTM-B:~$ cp ~/ArduinoApps/hatchagent-test
hatchagent-test/  hatchagent-test-src/
arduino@GTM-B:~$ cp -r ~/ArduinoApps/hatchagent-test-src/assets/. ~/A
AGENTS.md  ArduinoApps/
arduino@GTM-B:~$ cp -r ~/ArduinoApps/hatchagent-test-src/assets/. ~/ArduinoApps/hatchagent-test/a
app.yaml  assets/
arduino@GTM-B:~$ cp -r ~/ArduinoApps/hatchagent-test-src/assets/. ~/ArduinoApps/hatchagent-test/assets/
arduino@GTM-B:~$ ls ~/ArduinoApps/hatchagent-test/assets/
app.js  index.html  libs  style.css
arduino@GTM-B:~$ ls ~/ArduinoApps/hatchagent-test/assets/libs/
arduino.js  socket.io.min.js
arduino@GTM-B:~$
```

Figure 8. Repository clone and asset-copy commands

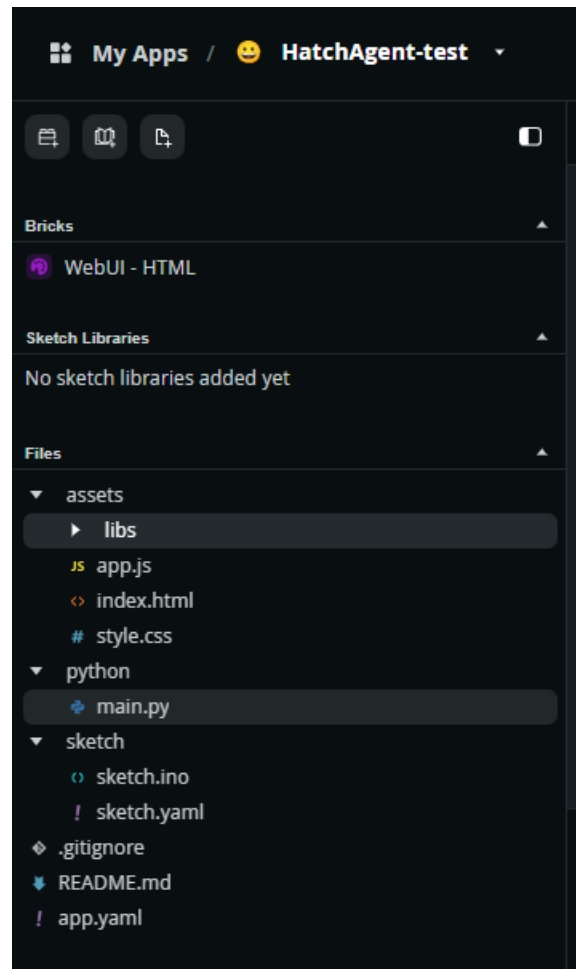


Figure 9. Expected HatchAgent project file tree

EXPECTED FILES

Confirm that assets contains index.html, app.js, style.css and the libs folder before continuing.

9. Add the agent logic

1. Open `python/main.py`.
2. Copy the full `main.py` from the downloaded repository.
3. Replace the blank Python example with the HatchAgent implementation.
4. Save the file and review the Python tab for startup errors when the project runs.

The supplied `main.py` serves the browser interface, sends user text to the local model, monitors sensor values and executes stored rules automatically.

Function area	Responsibility
Web serving	Provides the HatchAgent browser page
Model client	Sends supported prompts to the local inference server
Sensor monitoring	Reads supported values through Bridge calls
Rule engine	Evaluates conditions and triggers hardware effects

10. Install and start the local AI model

The model server runs beside the App Lab application as a separate process on the Linux host. Set it up from the App Lab terminal.

```
curl -L -o ~/setup.sh
https://raw.githubusercontent.com/hardikchadda/hatchagent/master/setup.sh

bash ~/setup.sh
```

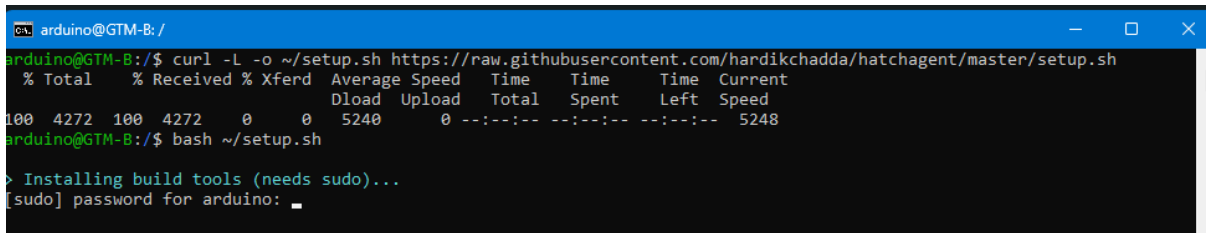


Figure 10. Running the HatchAgent setup script

- Installs the required llama.cpp build components.
- Downloads the HatchAgent model.
- Starts the model service on port 8080.

SEPARATE PROCESS

The App Lab application does not start the model server. Start the server before using the agent. After a reboot, run `bash ~/setup.sh` again; the supplied script is designed to skip completed build and download steps.

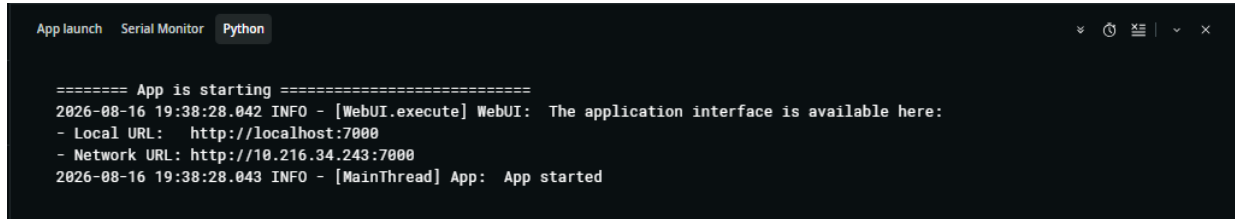
10.1 Recognise a stopped model server

Connection refused or url open error [Errno 111] indicates that the application cannot reach the local model service. Start the setup script again, then retest the interface.

11. Run and validate the agent

11.1 Start the application

1. Confirm the model server is running.
2. Select Run in Arduino App Lab.
3. Watch the build log while App Lab compiles and flashes the MCU sketch.
4. Open the Python tab and locate the printed network URL.
5. Open that URL in a browser and wait for the status badge to show on-device, ready.



```

App launch  Serial Monitor  Python

===== App is starting =====
2026-08-16 19:38:28.042 INFO - [WebUI.execute] WebUI: The application interface is available here:
- Local URL: http://localhost:7000
- Network URL: http://10.216.34.243:7000
2026-08-16 19:38:28.043 INFO - [MainThread] App: App started
  
```

Figure 11. Python log showing the application network URL

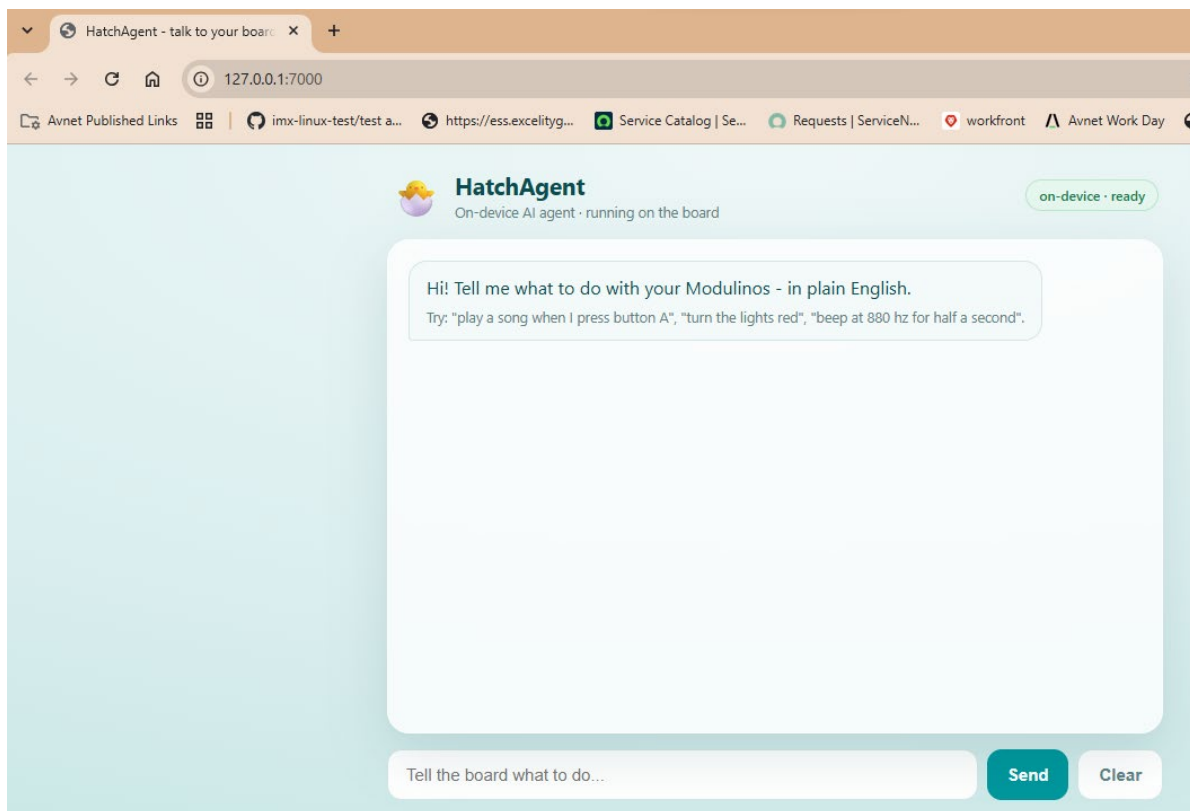


Figure 12. HatchAgent browser interface ready for a command

11.2 Verify connected capabilities

Run the source project's capability check in the terminal:

```
python3 -c "from arduino.app_utils import *; print('caps =', Bridge.call('caps'))"
```

The bitmask values are 1=Buttons, 2=Buzzer, 4=Pixels, 8=Knob, 16=Distance, 32=Movement and 64=Thermo. With all seven modules detected, the combined value is 127.

11.3 Test commands

You type	Expected response or action
turn the pixels blue	Pixels show solid blue
blink the pixels cyan, fast	Pixels blink cyan quickly
slowly breathe a soft purple	Pixels use a gentle purple pulse
if I press A, turn the pixels green	Pressing A turns the pixels green
if the knob goes above 60, turn red	Turning the knob above 60 turns the pixels red
when it's warm, pulse orange	Warming the Thermo sensor triggers an orange pulse
when something comes within 100 mm, flash red and beep	A nearby object triggers red flashing and a beep
what's the temperature?	The agent answers using the temperature reading

TEST METHOD

Start with direct output commands, then sensor queries, and finally conditional rules. This sequence helps isolate hardware, sensor and rule-engine issues.

12. Troubleshooting

Symptom	Likely area	Check
Board not visible	App Lab or network connection	Reconnect UNO Q and confirm the board is selected.
Build fails on libraries	sketch.yaml configuration	Confirm the required libraries and YAML formatting.
Distance value freezes	Distance library timing	Confirm setRangeTiming uses 100 as the second value in the library used by the build.
Browser page missing	Web UI or assets	Confirm the Brick, port, index.html, app.js, style.css and libs folder.
Connection refused / Errno 111	Model server stopped	Run <code>bash ~/setup.sh</code> and verify the server starts.
Capability value below 127	One or more modules not detected	Reseat Qwiic cables and rerun the app.
Direct command works but rule does not	Sensor value or rule condition	Check the sensor reading and use a threshold that can be reached during the test.
App works until reboot	Separate model process not restarted	Restart the model server after the board boots.

SUPPORT EVIDENCE

Record the exact error text, relevant build or Python log, selected project name, capability value and a photo of the Qwiic chain. Do not share passwords or private network credentials.

13. Fine-tuning and extensions

13.1 Optional model fine-tuning

The source project describes fine-tuning a compact Qwen2.5-1.5B model with LoRA using a synthetic dataset. The training objective is to map varied natural-language instructions to structured device programs rather than produce general conversational text.

- Use diverse phrasings that reflect how people naturally request the same action.
- Include direct commands, sensor questions and conditional rules.
- Keep the structured target output consistent.
- Separate training examples from evaluation examples.
- Test the resulting model with real spoken-style wording before deployment.

LORA BENEFITS IN THE SOURCE PROJECT

The source guide identifies faster training, lower GPU memory requirements, small adapter files and retention of base-model capabilities as benefits of LoRA.

13.2 Suggested extensions

Extension	Concept
More outputs	Map supported commands to displays, motors or relays with appropriate safety controls.
Rule persistence	Store validated rules so they can be restored after an application restart.
Device status	Add a dashboard view for capabilities, current sensor values and server status.
Controlled vocabulary	Add confirmations for actions where ambiguity could cause an unwanted physical response.
Evaluation suite	Create repeatable prompts and sensor scenarios to measure command and rule success.
Custom domain agent	Fine-tune examples for a specific permitted lab, education or automation workflow.

BUILD RESPONSIBLY

Before connecting higher-power or safety-critical hardware, assess false interpretations, safe states, authentication, physical limits and manual override behaviour.

14. Validation checklist, glossary and references

14.1 Acceptance checklist

Area	Acceptance check	Status
Hardware	All seven Modulinos are connected and detected	☐
MCU	Firmware builds and remote functions are available	☐
Distance	Sensor updates continuously after the timing fix	☐
Web UI	Browser interface loads from the printed network URL	☐
Assets	HTML, JavaScript, CSS and libraries are present	☐
MPU	Python application starts without an error	☐
Model	Local model service is running on port 8080	☐
Status	Interface displays on-device, ready	☐
Outputs	Pixels and buzzer respond to direct commands	☐
Sensors	Queries return current sensor information	☐
Rules	A conditional sensor rule triggers correctly	☐
Restart	Model server restart procedure has been tested	☐

14.2 Glossary

Term	Meaning
Agent	Application that interprets instructions and selects supported actions.
Bridge	Communication mechanism between the Linux and MCU domains.

Term	Meaning
Capability bitmask	Single integer whose bits represent detected hardware features.
GGUF	Model file format used by the local inference workflow.
llama.cpp	Local model inference runtime installed by the setup workflow.
LoRA	Low-Rank Adaptation method used for efficient fine-tuning.
MCU	Microcontroller domain responsible for deterministic hardware control.
MPU	Linux application domain responsible for Python, Web UI and AI workloads.
Qwiic	Keyed cable system used to connect the Modulino boards.
Rule engine	Logic that evaluates sensor conditions and triggers actions.

Conclusion

HatchAgent demonstrates an on-device, natural-language interface for embedded hardware on Arduino UNO Q. The design combines MCU-based sensor and actuator control with a Linux-hosted Python application, browser interface, rule engine and local language model. Once validated, the same architecture can be extended carefully to additional sensors, outputs and domain-specific command sets.

NEXT STEP

Complete the acceptance checklist with the supplied implementation before adding new commands, hardware or trained examples.

References used in this guide

- [HatchAgent source repository](#)
- [HatchAgent project ZIP](#)
- [HatchAgent setup script](#)