HIVIS MONITOR 2.0

Indoor Air Quality & Acoustic Monitoring System

Build Guide, Operating Manual & Project Report

Course	PROJ 108
Institution	Saskatchewan Polytechnic
Author	Ranieri dos Santos Vilela
Supervisor	Aaron Rodgers
Date	April 2026

Contents

Project Introduction

Title

HIVIS Monitor – Wearable Environmental Monitor for Industrial Workers

Overview

The HIVIS Monitor is a wearable IoT device designed to be worn on a worker's belt or integrated directly into a high-visibility safety vest. It continuously measures air quality, temperature, humidity, and noise levels, sending that data in real time to a self-hosted server. Workers can see their current readings on a small OLED display on the device, while supervisors and management can monitor the entire workforce through a live web dashboard from anywhere.

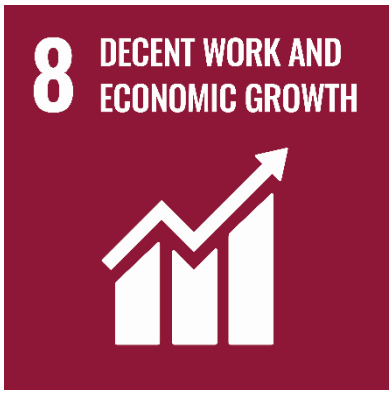
The primary target audience is tradespeople and industrial workers, along with their supervisors and safety managers who need visibility into field conditions.

Workers in trades and industrial environments are regularly exposed to conditions that can affect their health without knowing it — poor air quality, chemical vapors, elevated CO_2, and high noise levels are common in confined spaces, workshops, and industrial facilities. Most of the time, nobody is measuring any of it. Dedicated monitors exist, but they are expensive, require calibration, and are only deployed when regulations specifically demand it. The HIVIS Monitor fills that gap. It is not a certified gas detector and should not be treated as one — it is a continuous, real-time awareness tool that gives workers and employers data they currently have none of.

The system is fully self-hosted. No third-party cloud subscriptions, no data leaving your own infrastructure.

Purpose & SDG Alignment

SDG 3 — Good Health and Well-Being Provides continuous monitoring of environmental conditions that affect worker health — respiratory exposure, noise, and air quality — giving visibility into hazards that would otherwise go undetected.



SDG 8 — Decent Work and Economic Growth Safe working conditions are a fundamental part of decent work. The HIVIS Monitor gives workers and employers the data needed to identify risks before they turn into incidents or long-term health impacts.

SDG 9 — Industry, Innovation and Infrastructure Demonstrates that industrial safety monitoring does not have to rely on expensive proprietary systems. An open, self-hosted IoT solution built on accessible hardware is a practical path for workplaces that do not have enterprise budgets.

Required Skills and Knowledge

Before starting this build, you should be comfortable with basic soldering — through-hole and ideally surface-mount if you are assembling the PCB yourself. You will also need to know how to operate a 3D printer to print the enclosure. On the software side, basic computer skills are enough for most of the setup, but some familiarity with command line and SSH will be needed when configuring the server. No advanced programming knowledge is required — the firmware is provided ready to flash.

Materials and Tools

Hardware Bill of Materials

The hardware list below covers a single device. Reasoning for each major choice is recorded in Chapter 7.

Item	Part / Specification	Qty	CAD
MCU board	ESP32 DoIt DevKit V1 (4 MB flash)	1	$4.70
Air-quality sensor	Bosch BME688 breakout (I^2C)	1	$30.98
Microphone	InvenSense INMP441 MEMS (I^2S)	1	$8.20
Display	0.96" SSD1306 OLED, I^2C	1	$6.40
Buzzer	Passive piezo, 3.3 V tolerant	1	$1.20
Button	6 mm tactile through-hole	1	$0.30
Charger/Boost	J5019 1S Li-Ion charger + 5V boost	1	$6.00
Battery (Option A)	LiPo pouch 654060, 2100mAh, 3.7V	1	$25.00
Battery (Option B)	18650 Li-ion cell, 3.7V	1	$8.00
PCB	Custom 2-layer, 70 × 50 mm (JLCPCB)	1	$3
Enclosure	3D-printed PLA case (two parts)	1	$3
Misc	Resistors, Capacitors, JST Connector	-	$1
Total (Option A)			**~ $90**
Total (Option B)			**~ $73**

Server & Software

A single Linux host is sufficient for a small fleet (under ~50 devices). The reference deployment uses Docker Compose with seven containers: Mosquitto (MQTT broker), Node-RED (decoding and routing), InfluxDB 2.x (time-series storage), Grafana (dashboards), nginx (reverse proxy), certbot (Let's Encrypt), and an OTA file server.

Component	Version / Notes
OS	Ubuntu Server 22.04 LTS or Debian 12
Firmware toolchain	PlatformIO Core + Arduino-ESP32 framework
Broker	Mosquitto 2.0+ on port 8883 (TLS)
Database	InfluxDB 2.7+ with Flux query language
Dashboards	Grafana 10+ with InfluxDB datasource
VPN (optional)	Tailscale for remote admin

Tools

- Temperature-controlled soldering iron, 0.5 mm leaded or lead-free solder, flux pen.
- Digital multimeter for continuity and voltage checks.
- USB-C/micro USB cable rated for data (not power-only).
- 3D printer with 0.4 mm nozzle; PETG or PLA filament.
- Computer with VS Code + PlatformIO extension, plus a serial terminal.
- Wire Cutters / strippers
- Helping hands

Hardware Assembly

Circuit Diagram

The circuit diagram is bellow

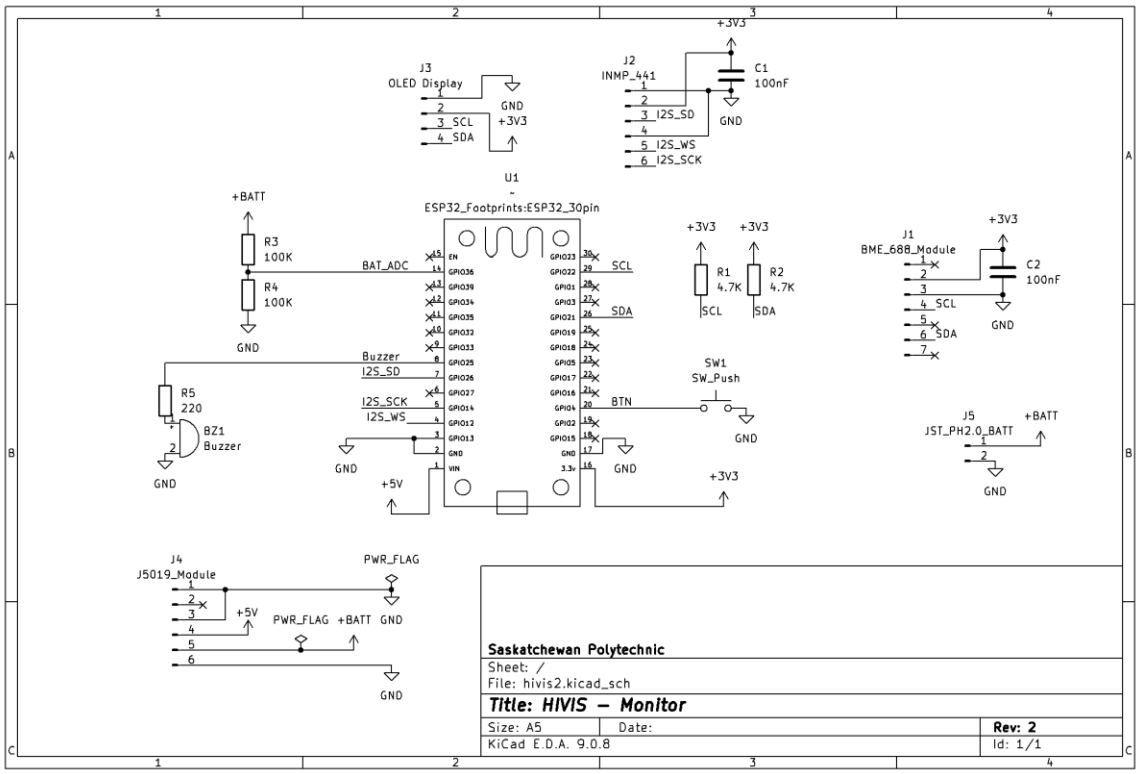


Figure 1 HIVIS Monitor Schematic (HIVIS2, Rev. 2)

Pin Assignments

All sensor connections are routed through the ESP32's hardware peripherals. I^2C is shared between the BME688 and the OLED; I^2S is dedicated to the microphone.

Function	ESP32 GPIO	Peripheral / Note
GPIO	Signal	Function
21	I2C SDA	I2C data — BME688 sensor + OLED display (shared bus)
22	I2C SCL	I2C clock — BME688 sensor + OLED display (shared bus)
12	I2S WS	INMP441 microphone — Word Select (L/R clock)
13	I2S LR	INMP441 microphone — L/R channel select (driven LOW → RIGHT channel)

14	I2S SCK	INMP441 microphone — Bit Clock
26	I2S SD	INMP441 microphone — Serial Data (audio in)
25	PWM (LEDC)	Buzzer — tone output via ledcAttachPin
4	Digital Input (PULLUP)	Button — active LOW, supports long-press & double-press
36	ADC (analog in)	Battery voltage — averaged 16-sample read through voltage divider (×2.2 ratio)

PCB Reference

The PCB is a 2-layer, 70 × 50 mm board with a top ground pour and routed signals on the bottom. Plated through-holes are 0.8 mm; the smallest clearance is 6 mil. Both renders below are produced from the project's gerber files.

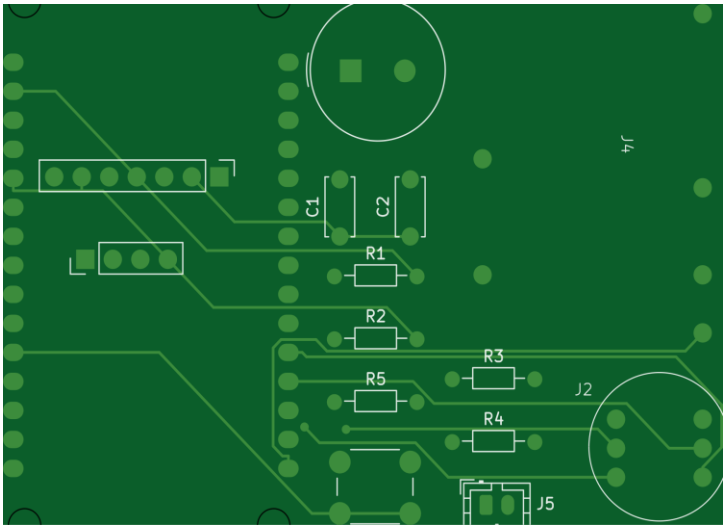


Figure 2 PCB top view showing component silkscreen, copper pour, and signal routing.

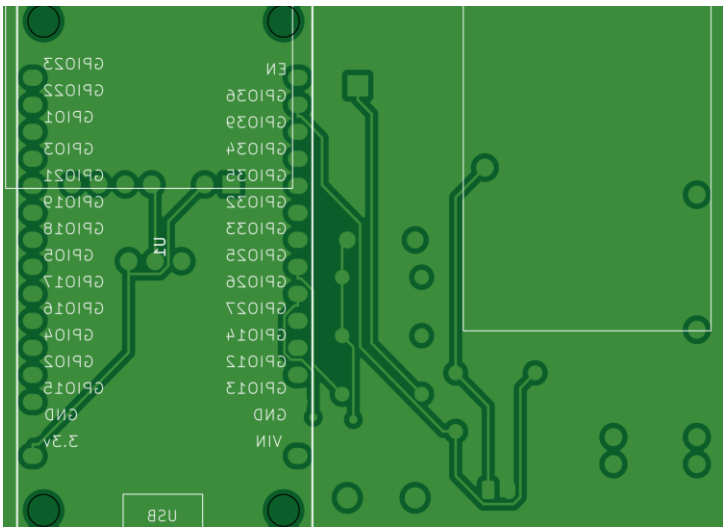


Figure 3 PCB bottom view; bottom-layer routing handles I^2S and indicator returns.

Assembly Steps

Step 1 — Inspect the Bare PCB

Before soldering anything, inspect the bare PCB under good lighting. Check that all copper traces are clean, there are no visible shorts between pads, and the silkscreen labels (R1–R5, C1–C2, J1–J5, BZ1, SW1) are all legible. If you ordered SMD assembly from JLCPCB, skip to Step 3 — the resistors, capacitors, and JST connector will already be soldered.

Step 2 — Solder SMD Components (if not pre-assembled)

If you are building from a bare PCB without assembly service, solder the SMD components first. These are the smallest parts and need to go on before anything else gets in the way.

Solder in this order:

- **C1, C2** — 100nF decoupling capacitors (0402)
- **R3, R4** — 100kΩ resistors, voltage divider for battery ADC (0402)
- **R1, R2** — 4.7kΩ pull-up resistors for I2C bus (0402)
- **R5** — 220Ω resistor for buzzer (0402)
- **J5** — JST-PH 2.0mm battery connector

Use flux, a fine tip, and work under magnification if possible. 0402 components are small — tweezers are essential. After soldering, check each pad under light for bridges.

Step 3 — Solder Through-Hole Components

With the SMD components done (or pre-assembled), move on to the through-hole parts. These are straightforward — insert from the front, solder from the back.

- **SW1** — Tactile button. Insert into the 4 pads and solder from the back. Press it a few times to confirm it clicks cleanly.
- **BZ1** — Passive piezo buzzer. Mind the polarity — the positive pin is marked on the silkscreen. Solder from the back.

Step 4 — Install the Breakout Modules

Now seat the breakout modules. All of these connect via pin headers — either solder header pins to the module first if they are not pre-installed, then solder them into the PCB.

- **BME688** — Top of the board. This is the environmental sensor. Seat it into J1, making sure VIN, GND, SCL, and SDA are aligned with the silkscreen. The sensor opening on the BME688 should face up and away from the PCB — do not cover it with anything.
- **SSD1306 OLED** — Left side, J3. 4-pin header: VCC, GND, SCL, SDA. The display should face outward so it is visible when the device is assembled in its enclosure.
- **INMP441 microphone** — J2. 6-pin connection: VDD, GND, SD, L/R, WS, SCK. The microphone port (small hole on the module) should face outward and not be blocked by other components or the enclosure.

Known PCB Issues — Rev. 2

In the current revision of the PCB, the footprint positions of the BME688 and INMP441 are incorrect.

- **BME688** — The pad layout does not align correctly with the breakout module. The workaround used in the current build is to connect the BME688 via short wires instead of soldering it directly to the PCB headers. It still functions correctly — just not seated directly on the board.
- **INMP441** — The footprint is mirrored. The workaround is to mount the module inverted (flipped upside down). It still connects and works correctly in this orientation.

Both issues are documented and will be corrected in the next PCB revision.

Step 5 — Install the ESP32

The ESP32 mounts on the **back side** of the PCB via two rows of pin headers. This keeps the USB port accessible from the back for programming and the antenna unobstructed.

Insert the pin headers into the PCB from the back, seat the ESP32 DevKit V1 onto the headers, and solder both sides — the headers to the PCB, and the ESP32 to the headers. Double-check alignment against the GPIO silkscreen on the PCB before soldering. Once soldered it is difficult to remove.

Figure 4 PCB Assembled Back

Step 6 — Install the J5019 Charger/Boost Module

The J5019 mounts at J4 and is soldered directly to the PCB. It handles both charging the battery from USB and boosting the battery voltage to 5V to power the ESP32. Solder all pads securely — this is the power path for the entire device, so cold joints here will cause problems.

The Micro-USB port on the J5019 is the charging input for the device. Make sure it is accessible after the board is installed in the enclosure.

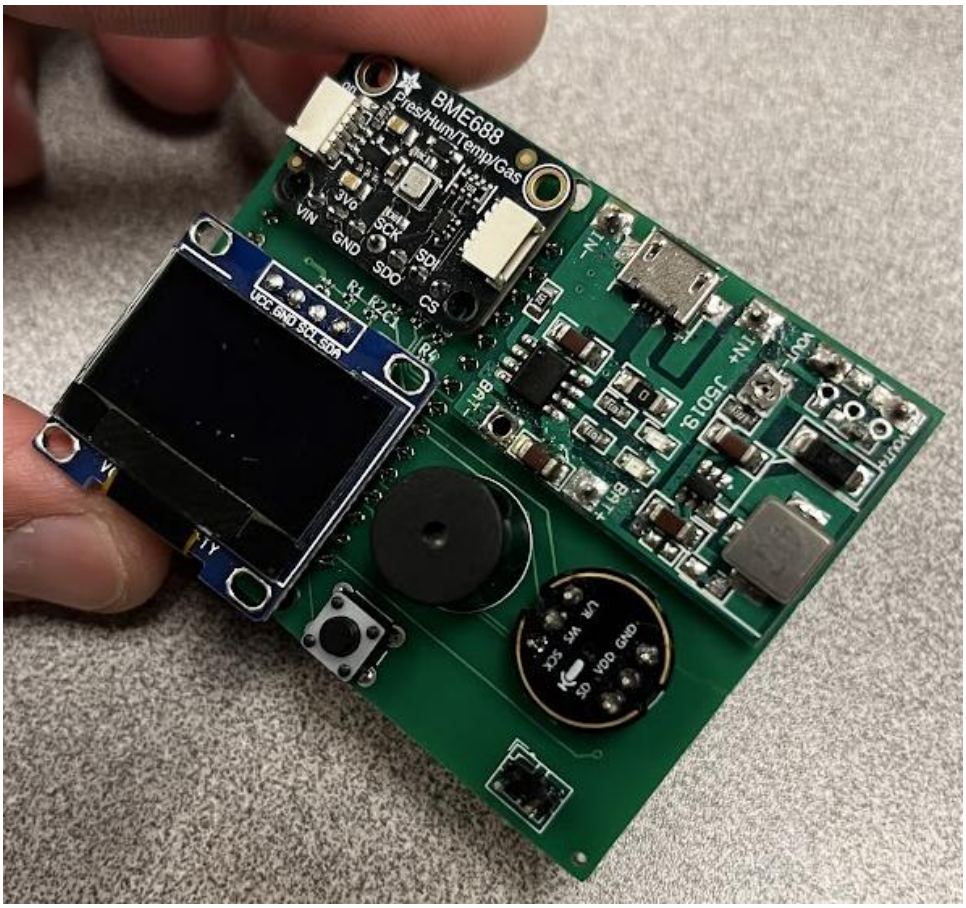


Figure 5 PCB Assembled Front

Step 7 — Connect the Battery

Connect the LiPo or 18650 battery to J5 via the JST-PH 2.0mm connector. **Check polarity before connecting** — red wire to positive (+BATT), black wire to GND. Reversing polarity will damage the J5019 and possibly the ESP32.

Once connected, the J5019 charge LED should indicate the battery state. If nothing powers on, check the JST connector orientation and verify the J5019 solder joints.

Step 8 — Mount the System in the Enclosure

The enclosure is a two-part 3D printed PLA case with a snap-fit lid. Print both parts with default infill and supports enabled. STL files are included with this submission.

Option A — LiPo Pouch Battery

Place the battery into the case first, sitting flat against the bottom. Then lower the PCB on top aligning the USB ports with the holes in the case, connect the JST battery cable before seating it fully. Close the lid and snap it into place.

Figure 6 Enclosure with pouch battery

Option B — 18650 Battery

Place the PCB into the case first align the USB ports with the holes in the case. The 18650 battery mounts on the outside of the enclosure — connect it to the J5019 module via the JST connector before closing.

Figure 7 Enclosure with External 18650 Battery

Once assembled, verify the following before closing the lid permanently:

- The OLED display is visible through the window in the lid
- The J5019 Micro-USB charging port is accessible from outside
- The ESP32 Micro-USB programming port is accessible from outside
- The button is aligned with its opening in the case

Snap the lid into place. The device is ready for use.

Troubleshooting Tips

Symptom	Likely Cause	Fix
Nothing powers on after battery connect	Reversed JST polarity or cold J5019 solder joint	Check polarity, reflow J5019 pads
OLED does not display anything	Bad header connection on J3	Reflow OLED header pins, check VCC/GND
BME688 not detected (I2C error in serial)	Header misaligned or cold joint on J1	Check SCL/SDA alignment, reflow
Microphone reads 0 dB constantly	INMP441 L/R pin not pulled LOW	Verify GPIO13 connection and R5
Button does not respond	SW1 solder bridge or wrong orientation	Check 4 pads, confirm click
Device not recognized by PC via USB	Power-only USB cable	Use a data-rated USB-C cable
ESP32 not detected	Cold joint on pin headers	Reflow all header pins on both sides

Software Setup

Development Environment

The firmware is written in C++ using the Arduino framework and built with PlatformIO. You will need the following installed on your computer:

- **Visual Studio Code** — download at code.visualstudio.com
- **PlatformIO extension** — install from the VS Code Extensions marketplace, search for "PlatformIO IDE"

That is all. PlatformIO handles the compiler, the Arduino-ESP32 framework, and all library dependencies automatically when you open the project for the first time. You do not need to install the Arduino IDE or any libraries manually.

Getting the Code

Clone or download the project repository from GitHub:

https://github.com/ranieriv/HIVIS2

Open the project folder in VS Code. PlatformIO will detect the platformio.ini file and begin downloading the required framework and libraries automatically. This may take a few minutes on the first run.

Firmware Architecture

The firmware is organized into modules, each responsible for a specific part of the device. Here is a quick overview of the key files:

File	Purpose
main.cpp	Boot sequence, main loop, ties all modules together
SensorModule.cpp	BME688 initialization, BSEC2 sensor reads, IAQ/CO2/VOC output
MicModule.cpp	INMP441 I2S microphone reads, dB SPL calculation
DisplayModule.cpp	SSD1306 OLED page rendering — IAQ, environment, system status
MqttModule.cpp	MQTT connection, publishing sensor data, registration flow
OTAModule.cpp	OTA update polling, firmware version check, HTTP update
ButtonModule.cpp	Single-button gesture detection — short, long, double, hold
BuzzerModule.cpp	Tone playback via LEDC PWM — distinct tones per event
ConfigHandler.cpp	LittleFS config.json read/write, NVS credential storage
OfflineBuffer.cpp	Offline data buffering to LittleFS, backfill on reconnect

The main loop runs every 3 seconds — it reads the sensors, updates the display, and publishes data to the MQTT broker. If WiFi or MQTT is unavailable, readings are saved to the offline buffer and uploaded automatically when the connection is restored.

Code Structure – Key Snippets

Button Gesture Detection

The device uses a single button (GPIO4) for all user interactions. Rather than separate buttons for each function, the firmware classifies each press by duration and timing. Short presses are held in a counter until the double-press window expires before being resolved — this is what allows a single button to distinguish a tap from a double-tap.

```
// On release: classify by hold duration
if (!_debounced && prevDebounced && _pressed) {
    _pressed = false;
    unsigned long duration = millis() - _pressStart;

    if (duration >= 5000)                              _factoryFlag = true;  // 5s →
factory reset
    else if (duration >= (unsigned long)_longPressMs) _longFlag    = true;  // long
press
    else {
        _pendingShorts++;                              // accumulate potential
double-press
        _lastReleaseTime = millis();
    }
}

// After the double-press window, resolve accumulated short presses
if (_pendingShorts > 0 && !_pressed &&
    millis() - _lastReleaseTime >= (unsigned long)_doublePressMs) {
    if (_pendingShorts >= 2) _doubleFlag = true;
    else                     _shortFlag  = true;
    _pendingShorts = 0;
}
```

Offline Buffer

When MQTT is unavailable, readings are not lost — they are saved to the device's LittleFS filesystem as NDJSON (one JSON object per line). If the buffer reaches its maximum of 500 records, the oldest entry is dropped to make room. When the connection is restored, all buffered readings are uploaded with a backfill: true flag so the server can distinguish them from live data.

```
void OfflineBuffer::store(const DisplayData& data) {
    int n = count();
    if (n >= MAX_RECORDS) _dropOldest();

    const char* mode = (n > 0) ? "a" : "w";
    File f = LittleFS.open(PATH, mode);
    if (!f) return;

    JsonDocument doc;
    doc["ts"]       = 0;
    doc["temp"]     = round(data.temp  * 10.0f) / 10.0f;
    doc["hum"]      = round(data.hum   * 10.0f) / 10.0f;
    doc["iaq"]      = round(data.iaq   * 10.0f) / 10.0f;
    doc["co2"]      = round(data.co2);
    doc["bvoc"]     = round(data.bvoc  * 100.0f) / 100.0f;
    doc["db"]       = round(data.db    * 10.0f)  / 10.0f;
    doc["backfill"] = true;

    serializeJson(doc, f);
    f.println();
    f.close();
}
```

BME 688 Sensor Read

The BSEC2 library handles sensor timing and processing internally. Each time it completes a sample, it fires a callback with the results. The firmware unpacks whichever outputs are ready and caches them. Calibration state is saved automatically when accuracy first reaches maximum or after a set time interval — this is what allows the device to skip the 30-minute warmup on subsequent boots.

```
void BME688Module::newDataCallback(const bme68xData data,
                                   const bsecOutputs outputs, Bsec2 bsec) {
    if (!outputs.nOutputs) return;
    for (uint8_t i = 0; i < outputs.nOutputs; i++) {
        const bsecData &o = outputs.output[i];
        switch (o.sensor_id) {
            case BSEC_OUTPUT_IAQ:                                  _iaq  = o.signal;
_accuracy = o.accuracy; break;
            case BSEC_OUTPUT_CO2_EQUIVALENT:                       _co2  = o.signal;
break;
            case BSEC_OUTPUT_BREATH_VOC_EQUIVALENT:                _bvoc = o.signal;
break;
            case BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_TEMPERATURE: _temp = o.signal;
break;
            case BSEC_OUTPUT_SENSOR_HEAT_COMPENSATED_HUMIDITY:    _hum  = o.signal;
break;
        }
    }
    bool accuracyJustHitMax = (_accuracy == 3 && lastSaveAcc != 3);
    bool intervalElapsed    = (millis() - lastSaveTime >= _saveIntervalMs);
    if (_accuracy > 0 && (accuracyJustHitMax || intervalElapsed))
        _instance->saveState();
}
```

The full source code for all modules is available at github.com/ranieriv/HIVIS2.

Device Boot Sequence

When the device powers on, it runs through the following steps in order:

1. Load config.json from LittleFS
2. Initialize hardware — I2C bus, OLED display, buzzer, button
3. Show splash screen, load saved credentials from NVS
4. If no WiFi or server credentials saved → launch captive portal (HIVIS-Setup-XXXX)
5. Initialize BME688 sensor and INMP441 microphone
6. Connect to WiFi (15 second timeout)
7. Sync time via NTP
8. If not provisioned → register with server via MQTT bootstrap
9. Connect to MQTT broker with provisioned credentials
10. Flush offline buffer if any records exist
11. Check for OTA firmware update (once every 24 hours)

12. Enter main loop — read sensors, update display, publish every 3 seconds

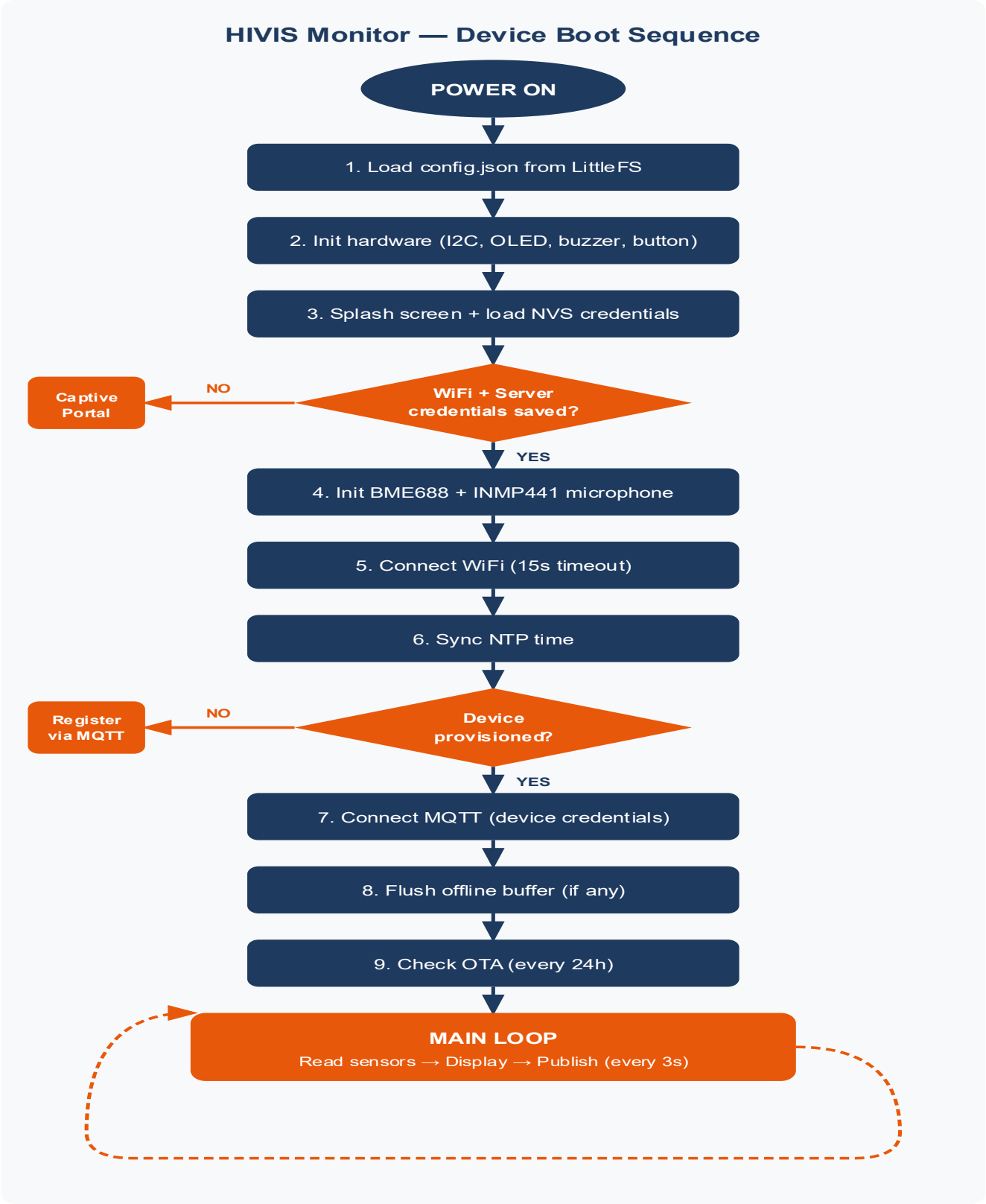


Figure 8 Device Boot Sequence

Flashing the Firmware

Flashing the device is done entirely inside VS Code through the PlatformIO sidebar. There are three steps and they must be done in order:

- **Step 1 — Build Filesystem Image** Click **Build Filesystem Image** in the PlatformIO project tasks. This packages the data/ folder — which contains config.json and other device files — into a LittleFS image ready to be flashed.
- **Step 2 — Upload Filesystem Image** Click **Upload Filesystem Image**. This flashes the LittleFS partition to the ESP32. Make sure the device is connected via USB before this step.
- **Step 3 — Upload Firmware** Click **Upload** (or **Upload and Monitor** if you want to see the serial output immediately after flashing). This compiles and flashes the main firmware to the device.

After flashing, the device will reboot and begin the boot sequence described above. If this is the first time the device has been flashed, it will launch the captive portal to collect WiFi and server credentials.

Configuration

Most device behaviour is controlled by config.json, stored in the data/ folder. You only need to edit this file if you want to change default settings — things like sensor read intervals, display timeout, pin assignments, or MQTT server address. For a standard build following this guide, the defaults work as-is.

After editing config.json, repeat Steps 1 and 2 (Build and Upload Filesystem Image) to push the changes to the device.

Library and Dependency Installation

All dependencies are declared in platformio.ini and downloaded automatically by PlatformIO when the project is opened. There is nothing to install manually. The key libraries used are:

Library	Purpose
BSEC2 (Bosch)	BME688 sensor — IAQ, eCO2, bVOC calculation
WiFiManager (tzapu)	Captive portal for WiFi provisioning
U8g2	SSD1306 OLED display rendering
ArduinoJson	JSON parsing for config and MQTT payloads
PubSubClient	MQTT client
LittleFS	Filesystem for config and offline buffer
Preferences	NVS credential storage

Troubleshooting Tips

Symptom	Likely Cause	Fix
PlatformIO fails to download libraries	No internet or firewall blocking	Check connection, try again
Upload fails — port not found	Wrong COM port or driver missing	Install CH340 driver, check Device Manager
Device launches captive portal every boot	NVS credentials not saved	Complete portal setup and save
bsec_state.bin does not exist on serial	LittleFS not uploaded	Run Upload Filesystem Image again
IAQ shows 0, accuracy = 0	BSEC2 still warming up	Wait ~30 minutes on first boot
Screen blank after flash	timeout_ms not set to 0 in config.json	Edit config.json, re-upload filesystem
MQTT state = -1 after boot	Broker unreachable or TLS error	Check server is running, check MQTT address in config
Offline buffer not clearing	MQTT connection not established	Check WiFi and broker, buffer uploads automatically on reconnect

Network Configuration

Connectivity Method

The HIVIS Monitor connects to the network over **WiFi (802.11 b/g/n)**. All communication between the device and the server uses two protocols:

- **MQTT over TLS (port 8883)** — for sensor data and device provisioning
- **HTTP (port 8090)** — for OTA firmware updates (LAN only)

There is no Bluetooth, cellular, or Ethernet. The device requires a 2.4GHz WiFi network to operate.

Wi-Fi Provisioning — First Boot

On first boot, the device has no WiFi credentials saved. It automatically launches a captive portal hotspot named **HIVIS-Setup-XXXX** (where XXXX is part of the device MAC address).

To configure the device:

1. On your phone or computer, connect to the **HIVIS-Setup-XXXX** WiFi network
2. A configuration page will open automatically (or browse to **192.168.4.1**)
3. Enter your WiFi network name (SSID) and password
4. Enter the MQTT server address (mqtt.hvht.net or your server IP)
5. Click Save

The device will reboot, connect to WiFi, and begin the registration process automatically. Credentials are stored securely in NVS (non-volatile storage) on the ESP32 and persist across reboots and power cycles.

To reconfigure WiFi at any time, long-press the button (GPIO 4) for more than 2 seconds. This relaunches the captive portal without wiping any other settings.

MQTT

MQTT is a lightweight messaging protocol designed for IoT devices. The HIVIS Monitor uses it to send sensor data to the server and to receive provisioning credentials on first registration.

All MQTT communication is secured with **TLS** — the connection is encrypted and the device verifies the server certificate before connecting. The broker runs on **port 8883**.

Topic Structure

Topic	Direction	Purpose
hivis/hivis-[mac]/data	Device → Server	Sensor data, published every 3 seconds
hivis/register	Device → Server	Registration request on first boot

hivis/provision/[mac]	Server → Device	Provisioning response — credentials delivered by server

Sensor Payload

Every 3 seconds the device publishes a JSON packet to its data topic:

```
{
  "device_id":   "hivis-4cc382c32764",
  "device_name": "Device 01",
  "ts":          1710518400,
  "ts_accurate": true,
  "fw_version":  "2.0.2",
  "temp":        22.5,
  "hum":         45.3,
  "iaq":         75.2,
  "co2":         412,
  "bvoc":        0.52,
  "db":          65.3,
  "bat_pct":     85,
  "bat_mv":      3850,
  "rssi":        -45,
  "accuracy":    3,
  "backfill":    false
}
```

The backfill flag is set to true when a reading was recorded offline and is being uploaded after reconnection. The accuracy field reflects the BSEC2 calibration state — values are only reliable at accuracy 1 or higher.

Device Provisioning Flow

Before a device can publish data, it must be registered with the server. This happens automatically on first boot and only needs to happen once.

```
Device (unprovisioned)
  | Connects using shared bootstrap credentials
  ▼
Mosquitto broker → Node-RED
  | Checks device MAC against whitelist
  | Generates unique credentials for this device
  | Adds device to Mosquitto access control
  | Publishes credentials back to device
  ▼
```

```
Device receives credentials → saves to NVS
  | Reconnects using its own unique credentials
  ▼
Device is now fully provisioned and publishing data
```

The device MAC address must be added to the server whitelist before registration is approved. See the section below on adding a new device.

Server Setup

The server stack runs entirely in Docker on a single Linux host. A low-power machine is sufficient for a small fleet — the reference deployment uses a Lenovo ThinkPad X200 running Ubuntu Server 22.04.

The stack consists of 7 containers managed by a single docker-compose.yml:

Container	Purpose	Port
Mosquitto	MQTT broker (TLS)	8883
Node-RED	Data routing and device provisioning	1880
InfluxDB 2.x	Time-series database	8086
Grafana	Dashboards	3000
OTA server	Firmware update delivery	8090
Nginx	Public website + InfluxDB proxy	80, 443
Certbot	SSL certificate auto-renewal	—

To start the server stack:

```
cd /opt/hivis
docker compose up -d
```

To check all containers are running:

```
docker compose ps
```

All 7 containers should show Up. If any are down:

```
docker compose up -d <service-name>
```

Full server configuration, Mosquitto TLS setup, Node-RED flows, and InfluxDB initialization are documented in the operations manual in the project repository.

Adding a New Device

Every device must be whitelisted on the server before it can register. The MAC address is printed to the serial monitor at boot.

Step 1 — Add the MAC to the Node-RED whitelist:

```
docker exec hivis-nodered nano /data/whitelist.json"
```

Add the device to the devices array:

```
{ "mac": "AA:BB:CC:DD:EE:FF", "authorized": true, "group":
"yourgroup", "notes": "Device description" }
```

Step 2 — Add the MAC to the OTA approved list:

```
ssh mqttadmin@172.16.1.156 "nano /opt/hivis/ota/devices.json"
```

Step 3 — Power on the device. It will auto-register and appear in Grafana within one minute.

Network Reference

The ports below are consistent across any deployment of this system. The IP addresses and domain names are specific to the reference deployment — yours will differ depending on your router and DNS setup.

Service	Address (reference deployment)	Port	Notes
Captive portal	192.168.4.1	—	Always this address — ESP32 default
MQTT broker	mqtt.hvht.net	8883	Your domain will differ
Node-RED	172.16.1.156	1880	LAN only — IP is deployment-specific
InfluxDB	172.16.1.156	8086	LAN only — IP is deployment-specific
Grafana	172.16.1.156	3000	LAN only — IP is deployment-specific
OTA server	172.16.1.156	8090	LAN only, restricted to local network
Public dashboard	hvht.net	443	Publicly accessible — your domain will differ

For remote server administration, this deployment uses **Tailscale** — a VPN that allows secure SSH access without exposing port 22 to the internet. Any similar VPN solution works; Tailscale is just what was used here.

Troubleshooting Tips

Symptom	Likely Cause	Fix
Device loops in captive portal	MQTT server address not saved	Complete portal setup, enter correct server address
MQTT state = -1 after boot	Broker unreachable or TLS error	Check Mosquitto container is running

Symptom	Likely Cause	Fix
Registration rejected	MAC not in whitelist	Add MAC to whitelist.json and retry
Device shows MQTT state = 5	Stale retained provisioning message	Erase NVS and reboot
No data in Grafana	Node-RED not routing to InfluxDB	Check Node-RED debug panel at port 1880
Captive portal not appearing	Device already has credentials	Long-press button > 2s to relaunch portal

Data Collection and Usage

Data Explanation

The HIVIS Monitor collects two categories of data — **environmental** and **system**.

Environmental data comes from two sensors:

The **Bosch BME688** measures temperature, humidity, and air quality. Using Bosch's BSEC2 algorithm, it produces an IAQ (Indoor Air Quality) score from 0 to 500, an estimated CO_2 equivalent (eCO_2) in ppm, and a breath VOC equivalent (bVOC) in ppm. These are calculated values, not direct measurements — the sensor detects a broad response to volatile organic compounds in the air and uses that alongside temperature, humidity, and pressure to estimate air quality. The sensor requires a burn-in period of approximately 30 minutes on first boot to begin producing reliable readings. Calibration state is saved automatically every 6 hours so subsequent boots do not require the full warmup.

The **INMP441 MEMS microphone** measures sound pressure level in dB SPL. It samples audio via the I2S interface and calculates a continuous noise level reading. It does not record audio — only the calculated dB value is stored and transmitted.

System data is collected from the ESP32 itself — battery percentage, battery voltage, WiFi signal strength (RSSI), firmware version, and BSEC2 accuracy level.

Data Format

Before transmission, all sensor readings are assembled into a single JSON packet every 3 seconds:

```
{
  "device_id":   "hivis-4cc382c32764",
  "device_name": "Device 01",
  "ts":          1710518400,
  "ts_accurate": true,
  "fw_version":  "2.0.2",
  "temp":        22.5,
  "hum":         45.3,
  "iaq":         75.2,
  "co2":         412,
  "bvoc":        0.52,
  "db":          65.3,
  "bat_pct":     85,
```

```
  "bat_mv":      3850,
  "rssi":        -45,
  "accuracy":    3,
  "backfill":    false
}
```

The ts field is a Unix timestamp synced via NTP. If the device has no internet connection at the time of recording, ts_accurate is set to false and an approximate timestamp is assigned when the data is uploaded. The backfill flag marks any reading that was stored offline and uploaded later.

On the server side, Node-RED receives the JSON packet, parses it, and writes it to InfluxDB 2.x as a time-series measurement. Each field becomes a separate InfluxDB field, and device_id, device_name, and group are stored as tags for filtering and grouping across multiple devices.

Data Transmission Protocols

The device uses two protocols:

MQTT over TLS (port 8883) is used for all sensor data and device provisioning. The device publishes a JSON payload to hivis/[device_id]/data every 3 seconds. If the connection drops, readings are saved locally to a LittleFS buffer (up to 500 records) and uploaded automatically when the connection is restored.

HTTP (port 8090) is used exclusively for OTA firmware updates. The device polls the OTA server once every 24 hours, checks if a newer firmware version is available, and applies the update automatically if one exists. This port is restricted to the local network and is not exposed publicly.

Platform Setup

The HIVIS Monitor is fully self-hosted — there is no third-party cloud service involved. The server running on a local Linux machine *is* the cloud. All data stays on your own infrastructure.

The server stack is deployed using Docker Compose and consists of Mosquitto, Node-RED, InfluxDB 2.x, Grafana, an OTA server, Nginx, and Certbot. The public dashboard at **hvht.net** is served by Nginx and queries InfluxDB through a read-only token — no data is sent to any external service.

All software in the stack is free and open source. There are no subscriptions, no API keys to purchase, and no data leaving your network unless you choose to expose it.

Full setup instructions are in the operations manual in the project repository.

Data Visualization

The system includes three Grafana dashboards, each serving a different purpose:

Fleet Overview refreshes every 30 seconds and shows the status of all active devices at a glance — a table with each device's latest IAQ, CO_2, noise, battery, and online status, plus a multi-device IAQ comparison chart and a battery level overview.

Per-Device refreshes every 5 seconds and shows a single device in real time — live gauges for IAQ, CO_2, noise, battery, and WiFi signal, plus time-series charts for all environmental and system metrics.

Historical refreshes every 5 minutes and covers long-term trends — IAQ, CO_2, temperature, humidity, noise, and battery drain over a selectable time range up to 7 days. Backfill data points are visually marked with a different annotation so they can be distinguished from real-time readings.

The public dashboard at **hvht.net** provides a simplified view accessible from any browser on any network, using a read-only InfluxDB token embedded in the page.

The three Grafana dashboards are included in the project repository as JSON provisioning files under server/grafana/provisioning/dashboards/. When the Docker stack starts, Grafana loads them automatically — no manual import needed. The repository also includes the datasource configuration and provider settings. Full details are in the project repository at github.com/ranieriv/HIVIS2. All dashboard configuration files and Flux queries are available in the project repository.

All APIs and software used in this system are free and open source. Grafana, InfluxDB, Mosquitto, and Node-RED are all open-source projects with no licensing fees. The InfluxDB query API uses Flux, an open query language. The public dashboard at hvht.net queries InfluxDB directly using a read-only token — no proprietary API or third-party service is involved.

Figure 9 Grafana Dashboard: Fleet Overview

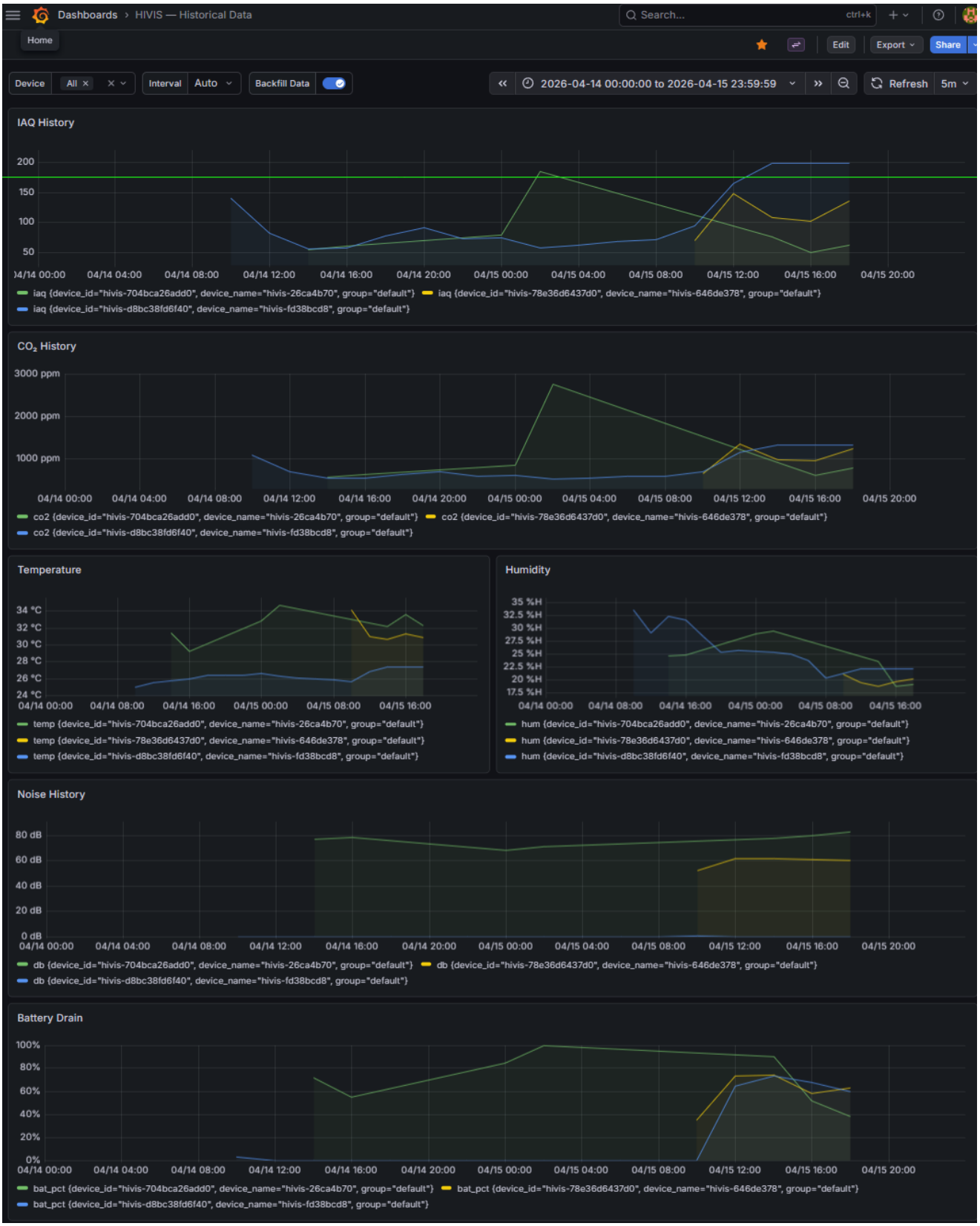


Figure 10 Grafana Dashboard: Historical Data

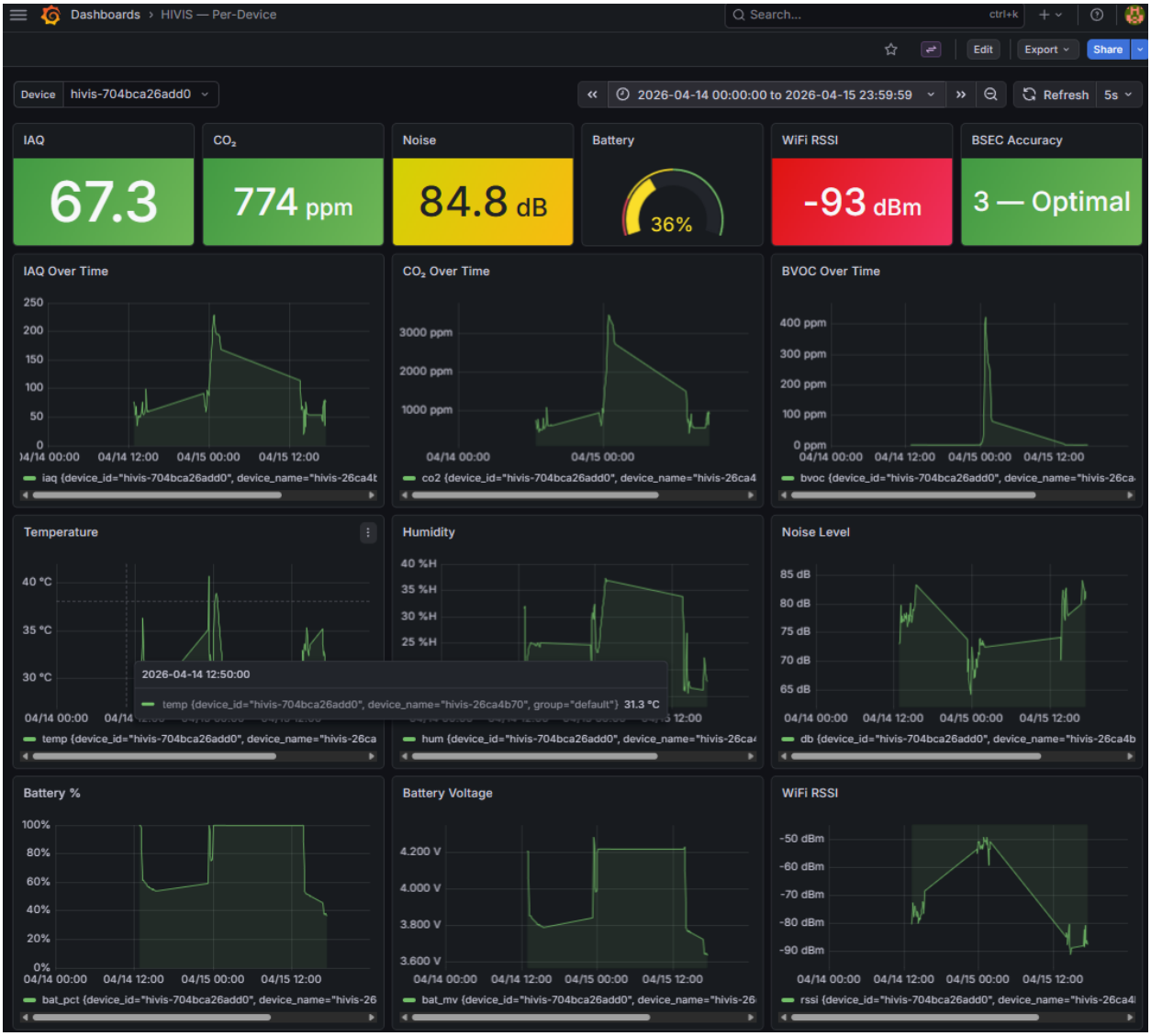


Figure 11 Grafana Dashboard: Per Device

Current Applications

The device was deployed at Saskatchewan Polytechnic during April 2026 as a real-world validation of the system. Three devices ran simultaneously across different locations, collecting continuous air quality and noise data over a 15-day period.

The results were significant. CO_2 levels exceeded the ASHRAE recommended threshold of 800 ppm on 9 out of 11 monitored days. The single worst event recorded was a CO_2 reading of 4,228 ppm and an IAQ score of 500 — the absolute sensor ceiling — on April 5 at 16:39 CST. On April 13, all three devices simultaneously recorded elevated readings, with one hitting 1,984 ppm CO_2 and an IAQ of 178 during occupied hours.

Before the HIVIS Monitor was deployed, none of this was visible to anyone in the building. Nobody knew those conditions existed. That is exactly the gap the device is designed to fill — not to replace certified

safety equipment, but to give people data they currently have none of. A worker or supervisor who can see a CO_2 spike in real time has a reason to act. Without that data, there is no reason to act at all.

Potential Applications

The current version of the device is a strong foundation, but there is clear room to expand. The most obvious next step is adding dedicated electrochemical sensors for specific gases — hydrogen sulfide (H_2S) and carbon monoxide (CO) are the most relevant for industrial and trades environments. The BME688 gives a general chemical exposure picture through its VOC response, but it cannot identify or quantify specific toxic gases. Dedicated sensors would make the device genuinely useful in environments where those specific hazards are present.

Beyond sensors, deploying more devices across a single worksite would allow spatial mapping of conditions — identifying which areas are consistently worse and giving management the data to make targeted interventions. The system already supports up to 50 devices on a single server with no architectural changes required.

Longer term, the historical data collected across a fleet of devices could be analyzed to identify patterns — times of day, weather conditions, occupancy levels — that correlate with poor air quality. That kind of analysis moves the system from reactive (something is bad right now) to predictive (this area tends to get bad on Friday afternoons), which is a much more powerful tool for worker safety management.

Additional Resources

Design Decisions

ESP32 The ESP32 has WiFi, Bluetooth, enough GPIO for every sensor in this build, costs under $5, has extensive documentation, and a massive community behind it. It also supports deep sleep modes that can bring power consumption down dramatically — not implemented in this version, but a real option for extending battery life in future iterations. An Arduino would need separate modules just to get online. A Raspberry Pi would be overkill — not just in processing power, but power consumption too, which matters a lot in a battery-powered wearable.

Bosch BME688 The BME688 measures temperature, humidity, pressure, and VOCs all in one chip. What made it interesting for this project is its AI capability — Bosch provides a tool called BME AI-Studio that lets you train the sensor to recognize specific gases. The original plan was to train it to detect H_2S. I didn't have time to do that in this iteration, but the hardware is ready for it. That's a clear path for the next version.

INMP441 It outputs digital audio directly over I2S, draws only 1.4 mA, and costs a few dollars. The ESP32 has hardware I2S support built in, so it just works. Easy to find, well-documented, no surprises.

SSD1306 OLED Display The worker needs to see their readings without pulling out a phone. A small display on the device makes that possible. The SSD1306 is cheap, low power, readable in different lighting conditions, and runs on the same I2C bus as the BME688 — no extra wiring needed.

J5019 Charger/Boost Module Charges the battery from USB and boosts the output to 5V in one module. It works with all LiPo batteries and 18650 cells, which is why the device supports both options. Cheapest combined solution I found for single-cell lithium batteries.

Project Website and Repository

Public Dashboard: hvht.net Live sensor data, updated every 10 seconds. Accessible from any browser on any network.

GitHub Repository: github.com/ranieriv/HIVIS2 Full firmware source code, PCB design files, server configuration, Docker Compose stack, operations manual, and system documentation. This is the primary reference for anyone building or maintaining the system.

Community Forums and Resources

DroneBot Workshop — Air Quality Monitoring dronebotworkshop.com/air-quality It covers a wide range of air quality sensors — MQ gas sensors, PM2.5 particulate sensors, and I2C environmental sensors — and shows how to use them with Arduino and ESP32. It was the foundation that led to exploring the BME family of sensors and eventually the BME688

PlatformIO Community community.platformio.org The official PlatformIO forum. Useful for build issues, library conflicts, and ESP32-specific firmware questions.

ESP32 Subreddit reddit.com/r/esp32 Active community for ESP32 hardware and firmware questions. Good for troubleshooting hardware issues and getting quick answers.

Bosch BSEC2 GitHub github.com/boschsensortec/BSEC-Arduino-library The official repository for the BSEC2 library used by the BME688 sensor. Useful for understanding calibration behaviour, accuracy levels, and library updates.

Further Development

The current version of the HIVIS Monitor is a working prototype that proves the concept. The most important next steps are adding dedicated electrochemical sensors for specific gases like H_2S and CO, which would move the device from a general awareness tool to something suitable for higher-risk industrial environments. Expanding the number of deployed devices and analyzing the historical data collected across a fleet are the other natural directions — the system already supports it, it just needs more devices in the field.