NATIONAL UNIVERSITY
OF SINGAPORE

OTTER

Dynamic Sensor Interfacing in Embedded System
Leveraging Large Language Models (LLMs)

Steven Waskito

FYP Dissertation

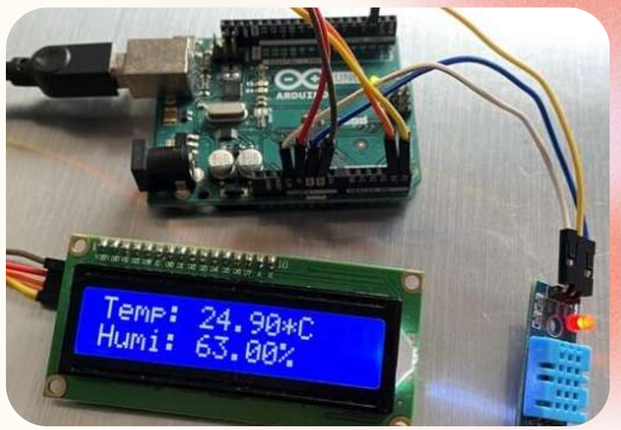
Prof. Ambuj Varshney

Overview

- Introduction & Background
- Problem Statement & Objective
- System Design
- Result & Evaluation
- Conclusion & Future Directions

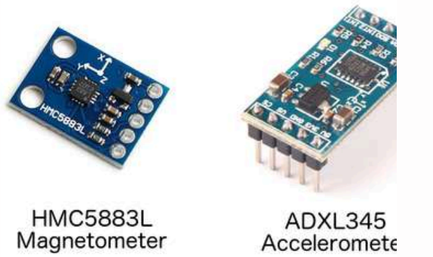
Introduction

What is Sensor Interfacing ?

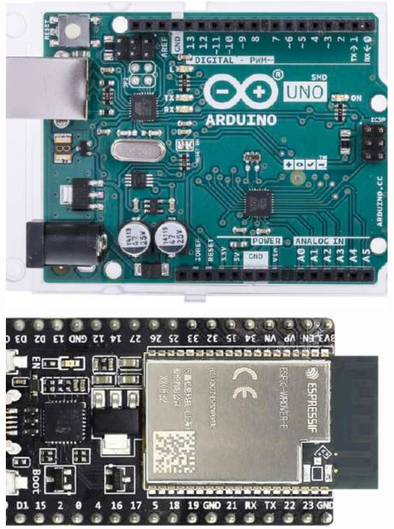


Temperature Humidity Sensor

Using Arduino UNO to interface with temperature and humidity i2c sensors



Sensors



Microcontroller

Background

Libraries & Embedded OS

Embedded sensor platforms traditionally require manual HAL driver development and firmware recompilation for any hardware change.

Manufacturer Open-Source Library

Manufacturer like Adafruit, SparkFun, and Pololu offers open-source high-level libraries. However, the offering is limited to their products.

TinyOS's NesC-based component-architecture

Improves modularity but compiles all drivers into a single, non-updatable firmware image.

Contiki OS and SOS

Supports dynamic module loading at the application level, yet cannot insert or update low-level sensor drivers at runtime.

Zephyr OS Device Tree

Enables dynamic peripheral binding but still demands precompiled driver modules, so true runtime plug-and-play remains out of reach.

Background

Scripting Language

Scripting language lowers the barrier of entry for well-known sensors. However, its bottleneck comes from new sensors, where datasheet reading is needed.

MicroPython’s Interactive Scripting

MicroPython enable interactive, hardware-agnostic I²C programming on microcontrollers without recompilation

Simplified Sensor API

Lowers the barrier to entry compared to traditional C++ development by providing simple APIs for sensor initialization and communication.

Manual Datasheet Parsing & No Runtime Modularity

Developers manually interpret datasheets and write register-level logic, and scripts lack true runtime modularity—adding or changing sensors requires halting execution and re-uploading the code.

Background

Generative AI

The generative AI workflow is not optimized for its capability. They are still limited on compile-flash-interface.

AI Code Generation and Code Assistants

AI solutions such as Cursor, CoPilot, and Ghostwriter let users describe sensor tasks and receive generated i2c sensor interface code directly

Automated Datasheet-to-Driver Generation

Supports dynamic module loading at the application level, yet cannot insert or update low-level sensor drivers at runtime.

LLM and RAG for Technical Document Reading

Enables dynamic peripheral binding but still demands precompiled driver modules, so true runtime plug-and-play remains out of reach.

The Gap

Precompiled Driver Constraint

Embedded OS sensor driver have to be precompiled.
Prevents on-the-fly integration

Compile, Flash, Execute

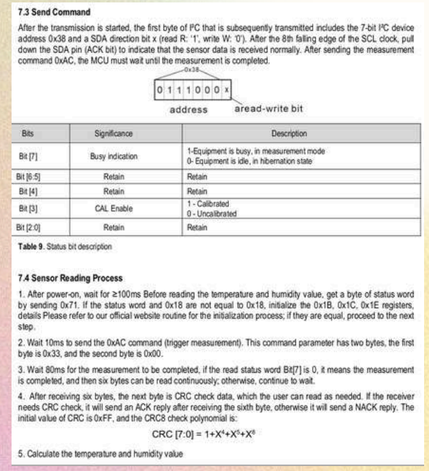
AI code generator rely on compile-flash-execute cycle.
Prevents rapid hardware changes

Static Sensor Interface

Adding, removing, or re-configuring sensor demand reflash of code

Sensor interfacing is a complex, slow, and inflexible process.

Sensor datasheets averaged **35+ pages** with inconsistent style between them



7.3 Send Command

After the transmission is started, the first byte of I²C that is subsequently transmitted includes the 7-bit I²C device address 0x38 and a SDA direction bit x (read R: '1', write W: '0'). After the 8th falling edge of the SCL clock, pull down the SDA pin (ACK bit) to indicate that the sensor data is received normally. After sending the measurement command 0xAC, the MCU must wait until the measurement is completed.

Bits	Significance	Description
Bit [7]	Busy indication	1-Equipment is busy, in measurement mode 0- Equipment is idle, in hibernation state
Bit [6:5]	Retain	Retain
Bit [4]	Retain	Retain
Bit [3]	CAL Enable	1 - Calibrated 0 - Uncalibrated
Bit [2:0]	Retain	Retain

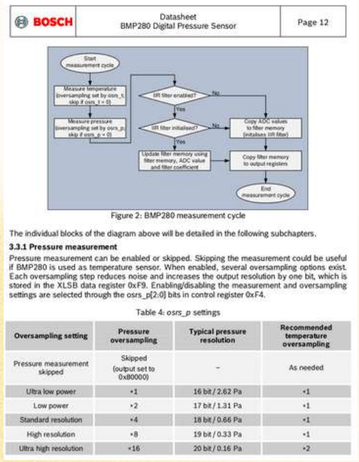
Table 9. Status bit description

7.4 Sensor Reading Process

1. After power-on, wait for ≥100ms Before reading the temperature and humidity value, get a byte of status word by sending 0x71. If the status word and 0x18 are not equal to 0x18, initialize the 0x1B, 0x1C, 0x1E registers, details Please refer to our official website routine for the initialization process; if they are equal, proceed to the next step.
2. Wait 10ms to send the 0xAC command (trigger measurement). This command parameter has two bytes, the first byte is 0x33, and the second byte is 0x00.
3. Wait 80ms for the measurement to be completed, if the read status word Bit[7] is 0, it means the measurement is completed, and then six bytes can be read continuously; otherwise, continue to wait.
4. After receiving six bytes, the next byte is CRC check data, which the user can read as needed. If the receiver needs CRC check, it will send an ACK reply after receiving the sixth byte, otherwise it will send a NACK reply. The initial value of CRC is 0xFF, and the CRC8 check polynomial is:

$$CRC\,[7:0] = 1+X^4+X^5+X^8$$

5. Calculate the temperature and humidity value



BOSCH | Datasheet BMP280 Digital Pressure Sensor | Page 12

Figure 2: BMP280 measurement cycle

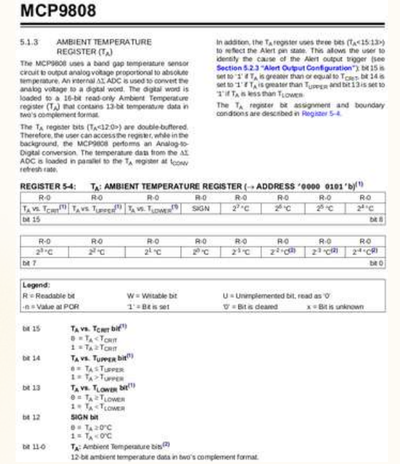
The individual blocks of the diagram above will be detailed in the following subchapters.

3.3.1 Pressure measurement

Pressure measurement can be enabled or skipped. Skipping the measurement could be useful if BMP280 is used as temperature sensor. When enabled, several oversampling options exist. Each oversampling step reduces noise and increases the output resolution by one bit, which is stored in the XLSB data register 0xF9. Enabling/disabling the measurement and oversampling settings are selected through the osrs_p[2:0] bits in control register 0xF4.

Table 4: *osrs_p* settings

Oversampling setting	Pressure oversampling	Typical pressure resolution	Recommended temperature oversampling
Pressure measurement skipped	Skipped (output set to 0x80000)	–	As needed
Ultra low power	×1	16 bit / 2.62 Pa	×1
Low power	×2	17 bit / 1.31 Pa	×1
Standard resolution	×4	18 bit / 0.66 Pa	×1
High resolution	×8	19 bit / 0.33 Pa	×1
Ultra high resolution	×16	20 bit / 0.16 Pa	×2



MCP9808

5.1.3 AMBIENT TEMPERATURE REGISTER (T_A)

The MCP9808 uses a band gap temperature sensor circuit to output analog voltage proportional to absolute temperature. An internal ΔΣ ADC is used to convert the analog voltage to a digital word. The digital word is loaded to a 16-bit read-only Ambient Temperature register (T_A) that contains 13-bit temperature data in two's complement format.

The T_A register bits (T_A<12:0>) are double-buffered. Therefore, the user can access the register, while in the background, the MCP9808 performs an Analog-to-Digital conversion. The temperature data from the ΔΣ ADC is loaded in parallel to the T_A register at t_{CONV} refresh rate.

In addition, the T_A register uses three bits (T_A<15:13>) to reflect the Alert pin state. This allows the user to identify the cause of the Alert output trigger (see **Section 5.2.3 "Alert Output Configuration"**). bit 15 is set to '1' if T_A is greater than or equal to T_{CRIT}, bit 14 is set to '1' if T_A is greater than T_{UPPER} and bit 13 is set to '1' if T_A is less than T_{LOWER}.

The T_A register bit assignment and boundary conditions are described in Register 5-4.

REGISTER 5-4: T_A: AMBIENT TEMPERATURE REGISTER (→ ADDRESS '0000 0101'b[1])

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
$T_A \geq T_{CRIT}$[1]	$T_A > T_{UPPER}$[1]	$T_A < T_{LOWER}$[1]	SIGN	2^7 °C	2^6 °C	2^5 °C	2^4 °C
bit 15							bit 8

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
2^3 °C	2^2 °C	2^1 °C	2^0 °C	2^{-1} °C	2^{-2} °C[2]	2^{-3} °C[2]	2^{-4} °C[2]
bit 7							bit 0

Legend:

R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared x = Bit is unknown

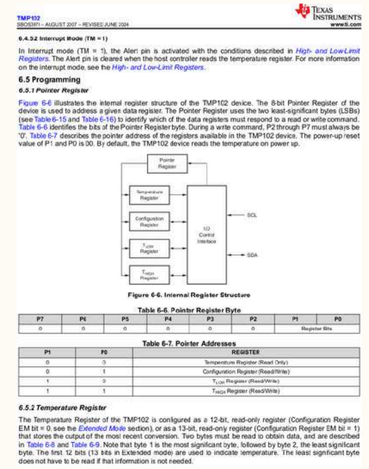
bit 15 **$T_A \geq T_{CRIT}$ bit**[1]
1 = $T_A \geq T_{CRIT}$
0 = $T_A < T_{CRIT}$

bit 14 **$T_A > T_{UPPER}$ bit**[1]
1 = $T_A > T_{UPPER}$
0 = $T_A \leq T_{UPPER}$

bit 13 **$T_A < T_{LOWER}$ bit**[1]
1 = $T_A < T_{LOWER}$
0 = $T_A \geq T_{LOWER}$

bit 12 **SIGN bit**
1 = $T_A < 0$°C
0 = $T_A \geq 0$°C

bit 11-0 **T_A**: Ambient Temperature bits[2]
12-bit ambient temperature data in two's complement format.



TMP102 SBOS397I – AUGUST 2007 – REVISED JUNE 2024 | TEXAS INSTRUMENTS www.ti.com

6.4.5.2 Interrupt Mode (TM = 1)

In Interrupt mode (TM = 1), the Alert pin is activated with the conditions described in *High- and Low-Limit Registers*. The Alert pin is cleared when the host controller reads the temperature register. For more information on the interrupt mode, see the *High- and Low-Limit Registers*.

6.5 Programming

6.5.1 Pointer Register

Figure 6-6 illustrates the internal register structure of the TMP102 device. The 8-bit Pointer Register of the device is used to address a given data register. The Pointer Register uses the two least-significant bytes (LSBs) (see Table 6-15 and Table 6-16) to identify which of the data registers must respond to a read or write command. Table 6-6 identifies the bits of the Pointer Register byte. During a write command, P2 through P7 must always be '0'. Table 6-7 describes the pointer address of the registers available in the TMP102 device. The power-up/reset value of P1 and P0 is 00. By default, the TMP102 device reads the temperature on power up.

Figure 6-6. Internal Register Structure

Table 6-6. Pointer Register Byte

P7	P6	P5	P4	P3	P2	P1	P0
0	0	0	0	0	0	Register Bits	

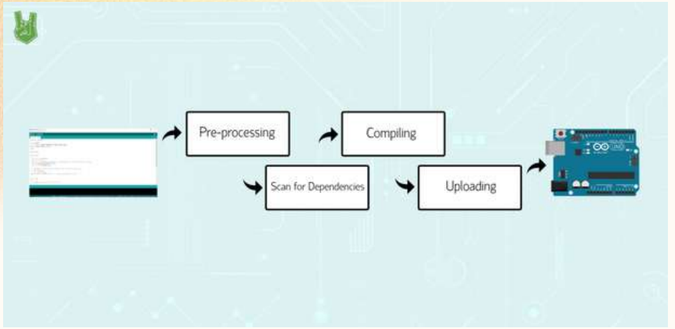
Table 6-7. Pointer Addresses

P1	P0	REGISTER
0	0	Temperature Register (Read Only)
0	1	Configuration Register (Read/Write)
1	0	T_{LOW} Register (Read/Write)
1	1	T_{HIGH} Register (Read/Write)

6.5.2 Temperature Register

The Temperature Register of the TMP102 is configured as a 12-bit, read-only register (Configuration Register EM bit = 0; see the *Extended Mode* section), or as a 13-bit, read-only register (Configuration Register EM bit = 1) that stores the output of the most recent conversion. Two bytes must be read to obtain data, and are described in Table 6-8 and Table 6-9. Note that byte 1 is the most significant byte, followed by byte 2, the least significant byte. The first 12 bits (13 bits in Extended mode) are used to indicate temperature. The least significant byte does not have to be read if that information is not needed.

Average firmware build + flash cycle is **30 seconds per attempt**



The Motivation

Eliminate manual driver development

Eliminate repeated firmware recompilation

Enable seamless runtime plug-prompt-and-play

Lower the barrier of entry for non-expert

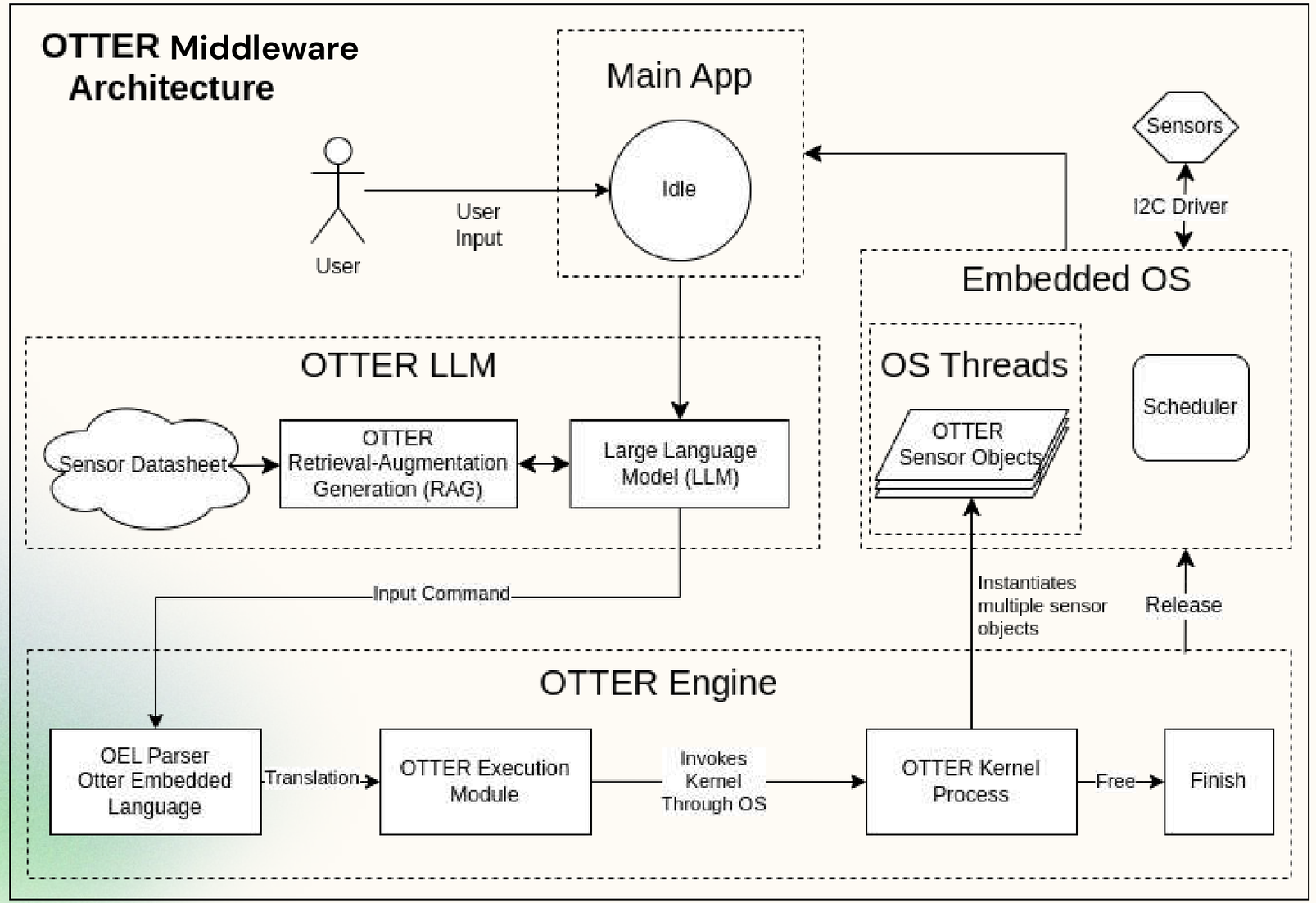
Accelerating prototyping and hardware iteration

Problem Statement

Is it possible to develop an
intelligent sensor interfacing system
that allows users to add and remove sensors through simple prompts, without manual coding or firmware recompilation?

System Design

The architecture was designed to incorporate **LLM intelligence**



Main App

The Main App serves as the **prompt input query.**

This is what the user will see when they are using OTTER OS

Figure 3.3: OTTER Main App Front-End

OTTER Embedded Language (OEL)

The template is designed to **support various sensors**

```
NEW_SENSOR = <SENSOR_NAME>;
PROTOCOL = i2c;
SENSOR_ADDR = <i2c_HEX_ADDRESS>;
INIT_CMD = <ARRAY_i2c_HEX_ADDRESS>;
NEW_SENSOR_READ;
READ_CMD = <ARRAY_i2c_HEX_ADDRESS>;
DATA_LEN = <INTEGER>;
DATA_KEY_VAL = <HASHMAP_INTEGER_STRING>;
DATA_FORMAT = <HASHMAP_INTEGER_ARRAYRANGE>;
SCALE_FORMAT = <HASHMAP_INTEGER_STRING>;
```

OTTER Embedded Language (OEL)

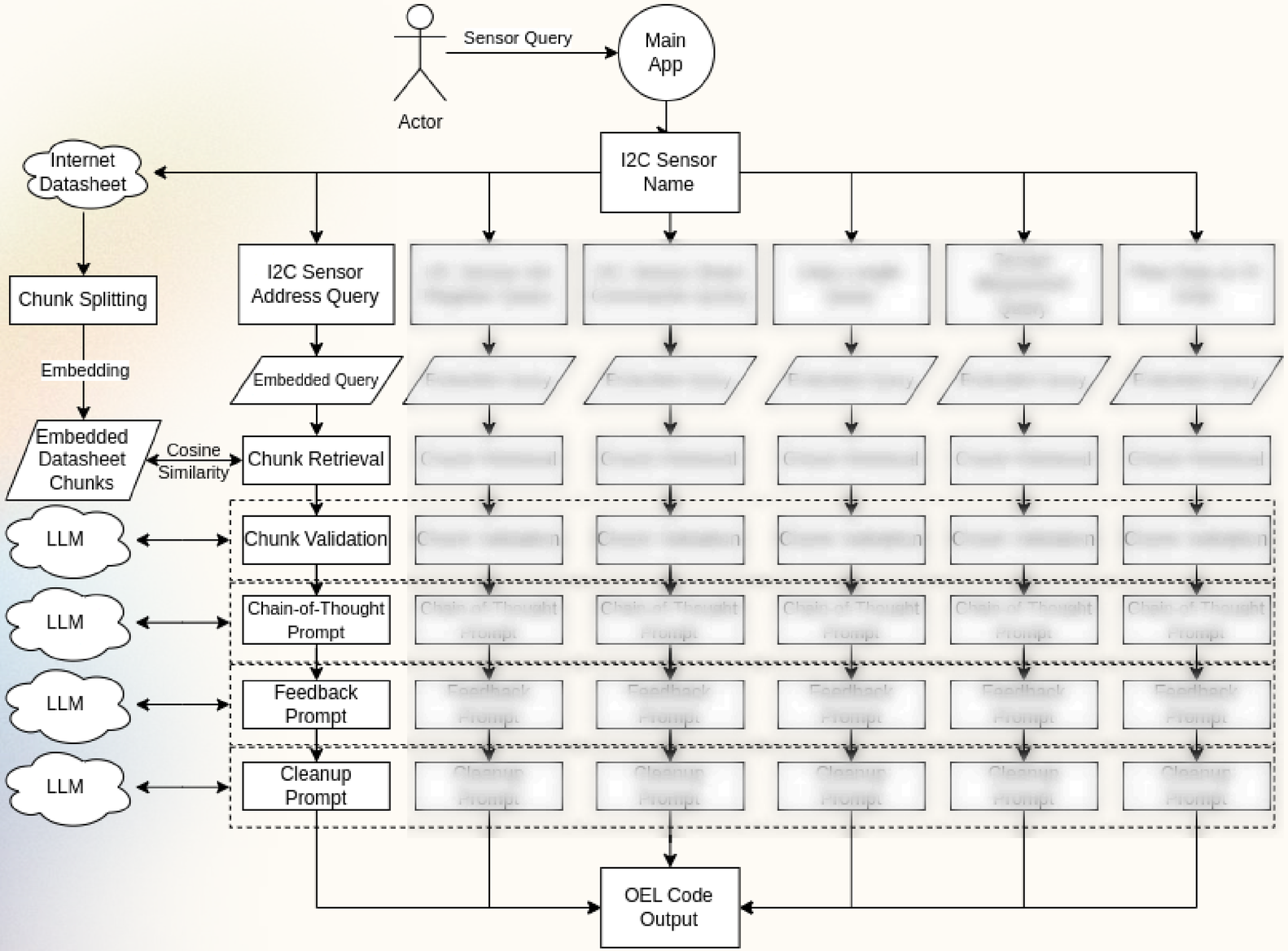
The OEL for
AHT20 Hum/Temp sensor

```
NEW_SENSOR = AHT20;
PROTOCOL = i2c;
SENSOR_ADDR = 0x38;
INIT_CMD = (0xBE, 0x08, 0x00);
NEW_SENSOR_READ;
READ_CMD = ();
DATA_LEN = 6;
DATA_KEY_VAL = (0: "HUM", 1: "TEMP");
DATA_FORMAT = (0: [12:31], 1: [28:47]);
SCALE_FORMAT = (0: "X 100.0 * 1048576.0 /", 1: "X 200.0 *
    1048576.0 / 50.0 -");
```

OTTER LLM

Consists of 3 major parts,

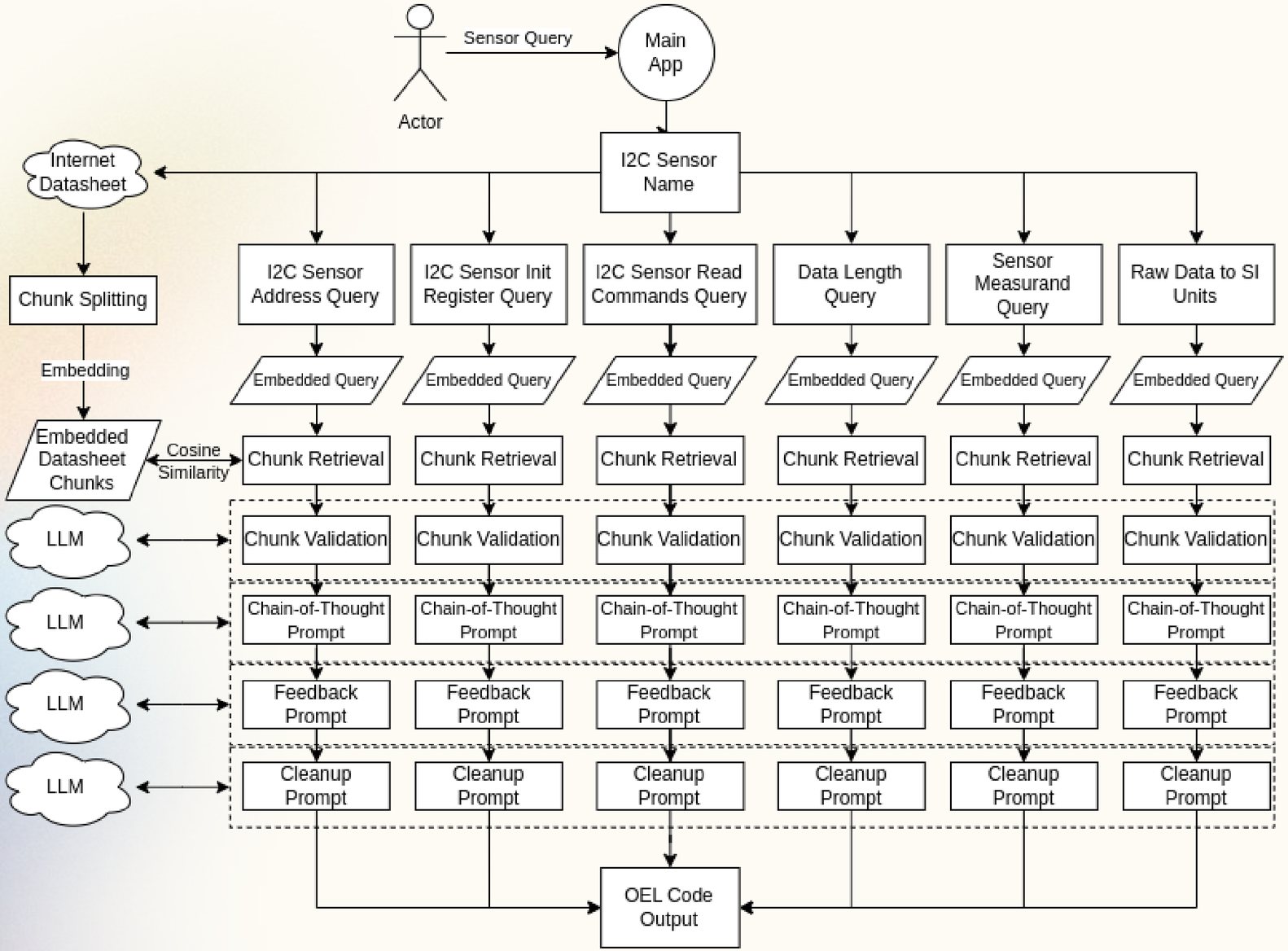
1. **Datasheet Processing**
2. **OEL Code Generation**
3. **OEL Code Assembly**



OTTER LLM

Consists of 3 major parts,

1. **Datasheet Processing**
2. **OEL Code Generation**
3. **OEL Code Assembly**



OTTER LLM Pipeline

Datasheet Processing

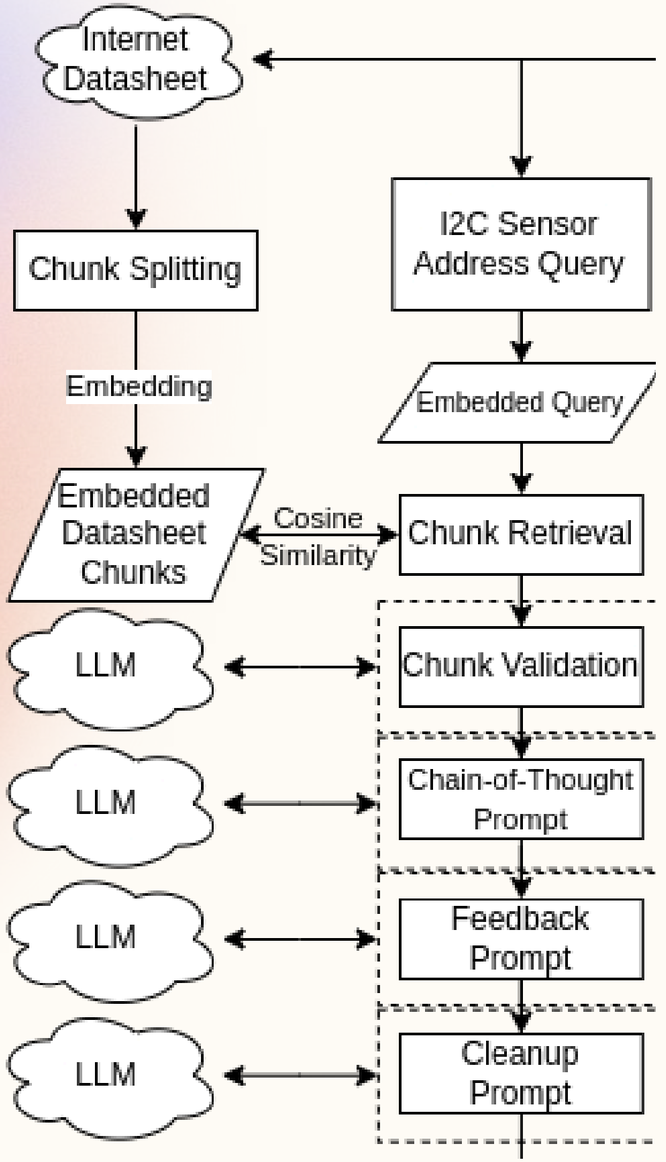
Automatically fetch the sensor's PDF, convert it to markdown, split into overlapping chunks, **embed each chunk (Ada-002)**, and index them in FAISS for retrieval.

OEL Code Generation

For each of the **six configuration** fields, run a **seven-step LLM pipeline**- query, embed, retrieve chunks, validate chunk, CoT prompt, feedback prompt, and cleanup prompt.

Final Assembly

Combine the outputs of all six query pipelines into a complete, syntactically correct OTTER Embedded Language specification ready for runtime execution.



OEL Code Generation

Initialize Query

Fill the template with the **sensor name** to ask
“What is the I²C address for AHT20?”

Query Embedding

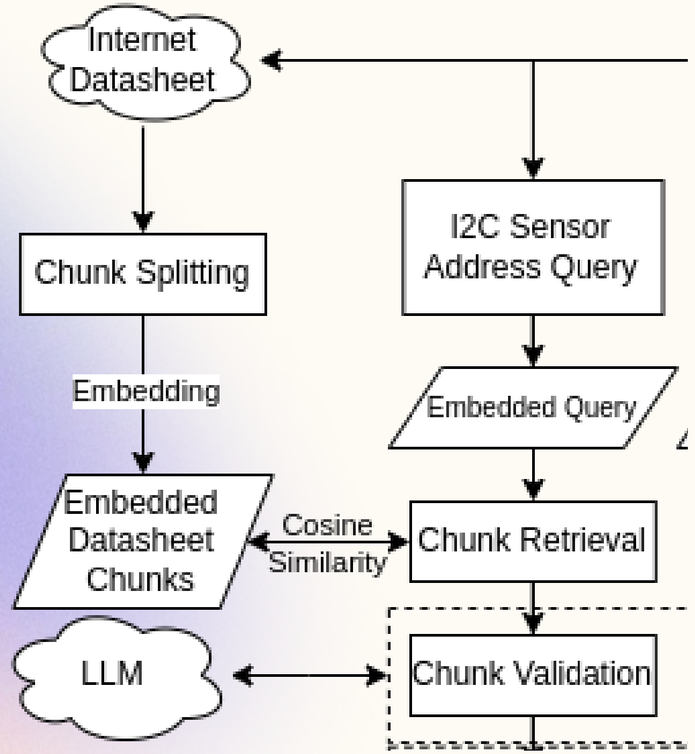
Vectorize the query using the Ada-002 embedding model.

Chunk Retrieval

Use FAISS **cosine similarity** to retrieve the top 3 datasheet chunks by cosine similarity to the query vector.

Chunk Validation

Ask the LLM if each **chunk is helpful** and discard any irrelevant ones.
“Is this chunk helpful for I2C Sensor Address Query?”



OEL Code Generation

RAG + Chain-of-Thought (CoT) Prompting

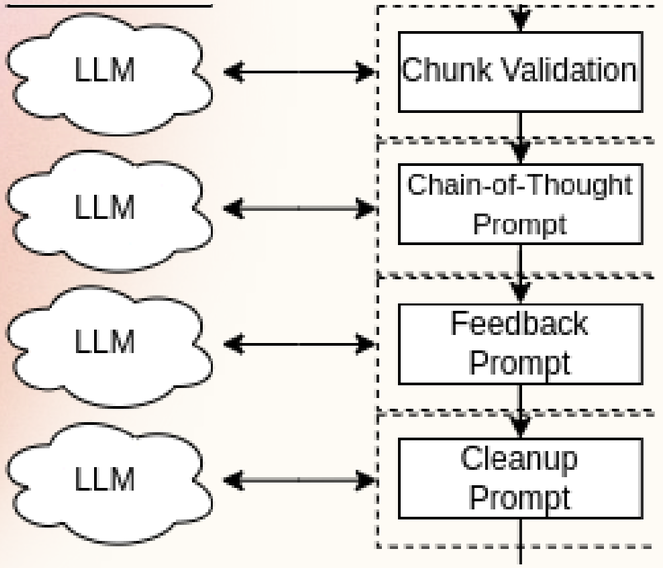
Augment the validated datasheet chunk with the query.
Explicitly ask the LLM to do **step-by-step reasoning**.

Feedback Prompt (Step-back Prompt)

Wrap the CoT output in “My expert told me: …”
Explicitly ask the LLM to **extract the concise component answer.**
“Do I need to read specific registers? If not, say "INOP"
Finish the sentence: The hexadecimal values are: “

Cleanup Prompt

Re-prompt with formatting rules to **produce a clean output** of the OEL component needed.
“Extract only the hexadecimal values as individual 1-byte entries, separated by comma. Start with 0x…”



OTTER Engine

OEL Parser

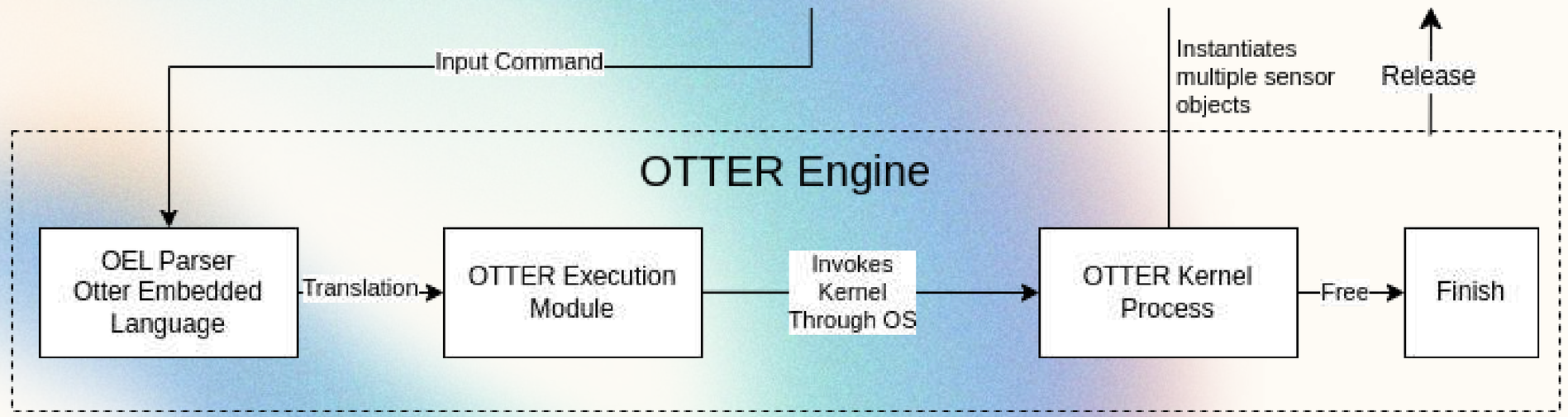
Parses the OEL String to each **individual OEL components** and deterministically for the creation of SensorConfig struct.

Execution Module

Poll serial non-blocking for OEL commands, parse and validate them into Sensor objects, confirm, and **delegate creation or teardown to the kernel.**

Kernel Process

Instantiates the Sensor object in the OS from the object provided by the execution module. Safely release and free the Sensor object on termination or removal of sensors.



Embedded OS

OS Agnostic

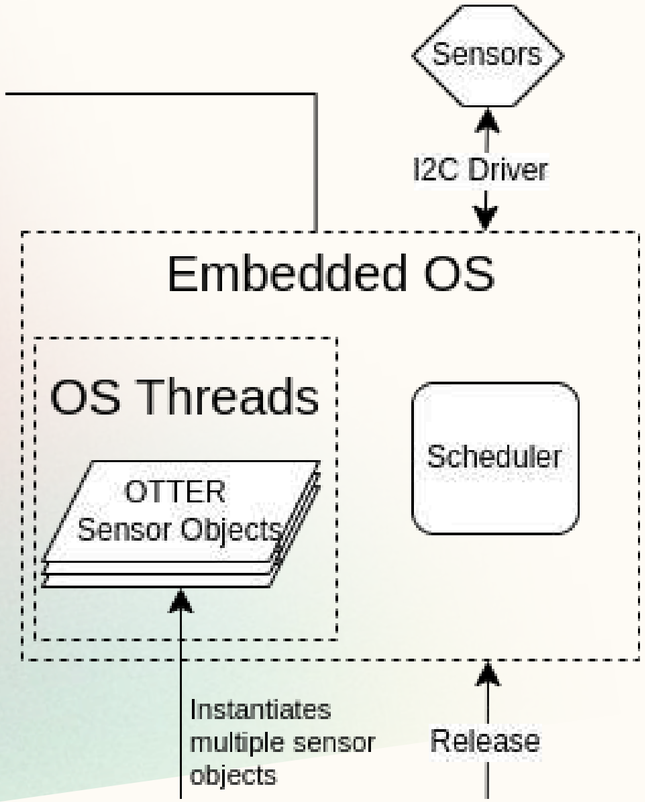
Use any OS with **i2c library** and **multithreading** to dynamically spawn sensor threads, schedule them, and share i2c bus access via mutex.

Deterministic and Concurrent Threads

Each OTTER Sensor Object runs in its **own thread** to perform i2c reads. The thread, running in parallel, handles data extraction and scaling.

Fault Isolation and Scalability

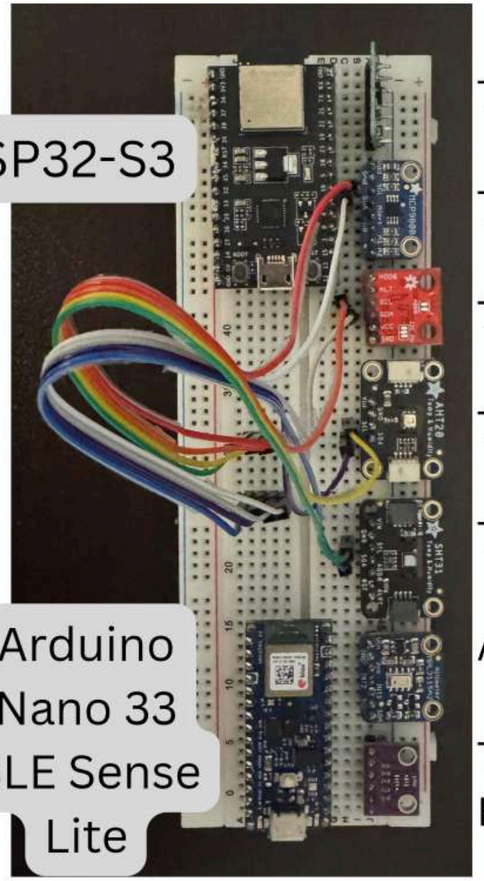
Thread-local execution **isolates sensor failures**, and OS scheduling ensures data integrity, **modularity**, and plug-and-play adaptability.



Experiment and Evaluation

Experiment Hardware

Arduino and Sensors



- Time of Flight Sensor (irs11a 0j9776)
- Temperature Sensor (MCP9808)
- Temperature Sensor (TMP102)
- Temperature & Humidity Sensor (AHT20)
- Temperature & Humidity Sensor (SHT31)
- Altitude/Pressure Sensor (MPL3115A2)
- Temperature & Barometric Sensor (GY-BMP280)

Prompting Strategies

Overview

Through the empirical observation from various different prompts, we have compiled a list of prompt strategies that seem to be effective

A good whitepaper was published by Google shortly after the deadline of the dissertation.

System Prompt

The first **directive** on any query, the system prompt is crucial for assigning **roles**. “You are a helpful assistant and expert in I2C Sensors.

Pronouns

Using pronouns such as **“You”, “I”, and “My Expert”** is important to give directives, ask questions, or give additional information.

Explicit Negative Output

When the answer to the question is “no” or “don’t need”, the LLM will give a long explanation instead of giving negative output. Therefore, we must ask the LLM a yes or no question and give **“if no, say “NO””**

Finish the Sentence

A good way to direct LLM is to **ask it to finish a sentence**. A prompt “Finish this sentence: The hexadecimal values are: “ performs better than questioning the LLM “What are the hexadecimal values?”

Rules

For more complex components, we found that giving a list of rules of **what to do** and **what NOT to do** is helpful to prevent LLM to veer off.

OTTER Embedded Language (OEL)

Sensors Datasheet are Unique

On the table, we observe the differences in i2c sensors measurement range, resolutions, and datasheet

Complexity

A naive view is the longer the datasheet, the more complex the sensor is. We shall test whether OEL is able to handle sensor and datasheet with **various complexity**

Sensor Compatibility Experiment.

i^2c Sensors	Measurement Range	Resolution	Datasheet Length
TMP102	-40°C to 125°C (Temp)	0.0625°C	33 pages
MCP9808	-40°C to 125°C (Temp)	0.0625°C	52 pages
AHT20	-40°C to 85°C (Temp/Hum)	0.01°C / 0.024%RH	16 pages
SHT31	-40°C to 125°C (Temp/Hum)	0.01°C / 0.01%RH	22 pages
MPL3115A2	20 kPa to 110 kPa (Pressure)	0.25 Pa (Altimeter)	51 pages
VL53L0X	30 mm to 2 m (Distance)	1 mm	38 pages

Table 3.1: Comparison of selected i^2c sensors and their datasheet lengths. All humidity measurement ranges from 0-100%RH

OTTER Embedded Language (OEL)

A working OEL produces a correct sensor interface

4/6 Sensors ran with the correct output
(AHT20, MCP9808, SHT31, and TMP102)

```
===== Sensor Configuration =====
Sensor Name: MCP9808
Protocol: I2C
I2C Address: 0x18
Wakeup Command (0 bytes):
Read Command (1 bytes): 0x05
Incoming Data Length: 2
Data Keys and Formats:
  0: "TEMP" | Bits [0:15] | Scale: x 4096 % 0.0625 *
================================
Running Sensor Thread: MCP9808
TEMP = 25.44
Running Sensor Thread: MCP9808
TEMP = 25.44
Running Sensor Thread: MCP9808
TEMP = 25.44
Running Sensor Thread: MCP9808
TEMP = 25.44
```

Figure 4.1: MCP9808 Sensor Struct and Output

OTTER End-to-End

Evaluation of 40 trials

Prompt → Datasheet Retrieval → OEL generation → runtime → I²C output

80.5% Accuracy

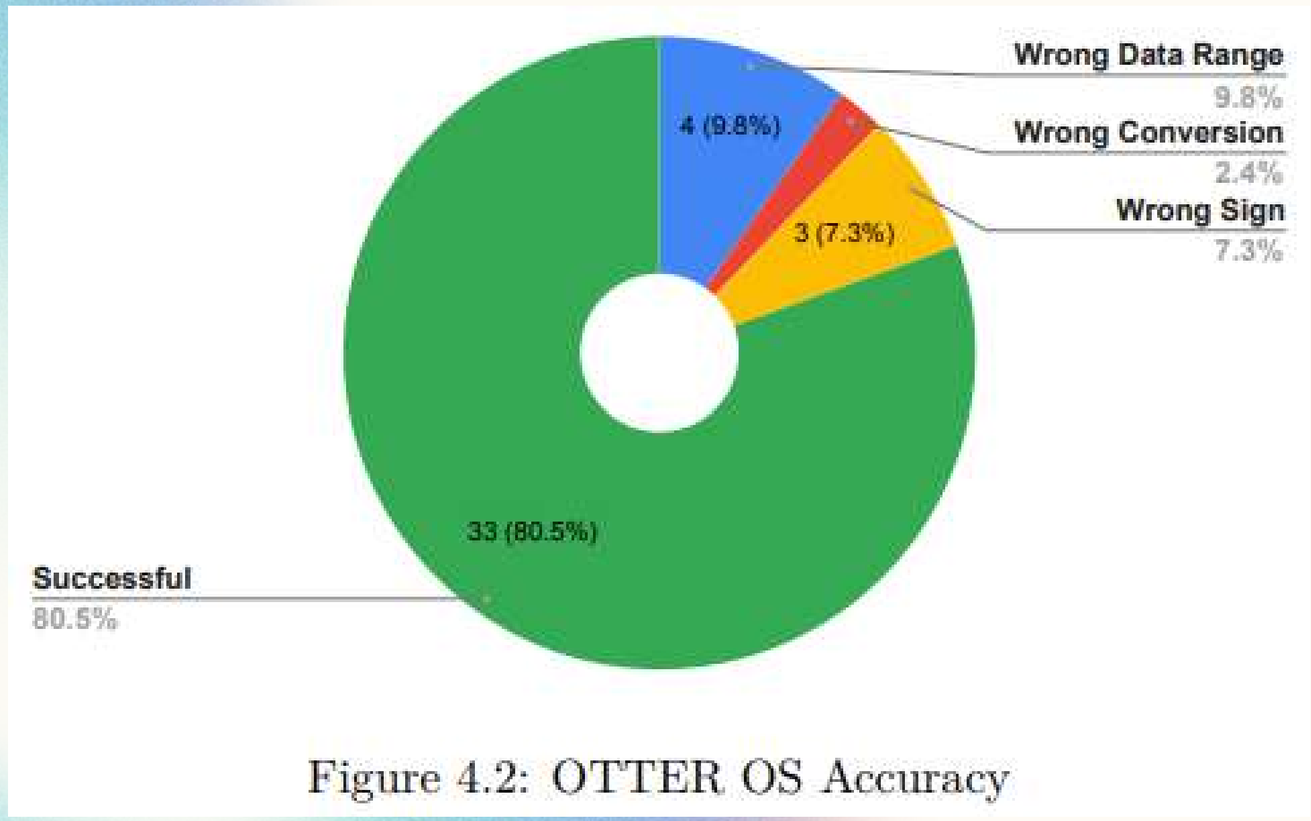


Figure 4.2: OTTER OS Accuracy

OTTER LLM Retrieval Augmentation Generation (RAG)

Evaluation of 40 trials

FAISS Retrieval Accuracy 75%

Relying on cosine similarity, the FAISS retrieval is subject to embedding accuracy, which is **not designed for technical information**.

LLM Validation Accuracy 94.9%

Filters out vectorially similar but functionally irrelevant sections, ensuring only genuinely **useful datasheet content** is passed

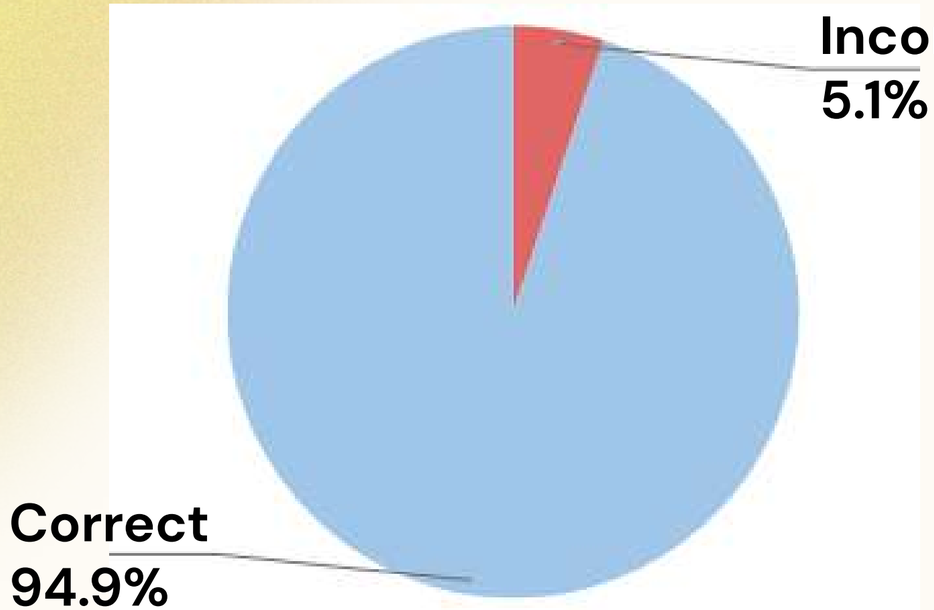


Figure 4.3: Chunk Validation Accuracy

Multiple Sensor Hot-Swap Plug-Prompt-and-Play

Evaluation

Concurrent Multi-Sensor Support

Successfully ran four I²C devices on a single bus with each sensor thread polling independently and producing continuous, accurate readings **without conflict**.

```
Running Sensor Thread: AHT20
HUM = 57.66
TEMP = 24.32
Running Sensor Thread: SHT31
TEMP = 24.55
HUM = 54.42
Running Sensor Thread: TMP102
TEMP = 24.37
Running Sensor Thread: MCP9808
TEMP = 24.94
```

(AHT20, SHT31, TMP102, MCP9808) Concurrent Sensor Reading

Multiple Sensor Hot-Swap Plug-Prompt-and-Play

Evaluation

Dynamic Plug-Prompt-and-Play

Demonstrated runtime addition and removal via NEW_SENSOR / SENSOR_REMOVE commands (e.g., TMP102 added→removed→readded) **without firmware recompilation.**

Threads Lifecycle and Hot-Swaps

OTTER Engine **safely spawns and terminates sensor** threads, handling memory, I²C initialization, and deallocation, to enable seamless hot-swapping and hardware reconfiguration.

```
Running Sensor Thread: TMP102
Temperature = 23.62
Freeing thread
Removed Sensor: TMP102
Returning to OS
===== Sensor Configuration =====
Sensor Name: TMP102
Protocol: I2C
I2C Address: 0x48
Wakeup Command (0 bytes):
Read Command (0 bytes):
Incoming Data Length: 2
Data Keys and Formats:
  0: "Temperature" | Bits [0:11] | Scale: X 2048 / 2 % 4096 * X - 0.0625
================================
Running Sensor Thread: TMP102
Temperature = 23.62
```

Figure 4.5: Plug-and-Play Sensor Interface with TMP102

Latency
Evaluation of 40 trials

End-to-End Latency

Median: 202.035 s

Mean: 200.090 s

Interquartile range: 182.131 s–216.731 s

LLM Component Latency

Chunk retrieval (FAISS): 0.786 s mean

Chunk validation (LLM): 10.067 s mean

OTTER LLM Component:

- Sensor address: 15.230 s
- Initialization command: 31.464 s
- Read command: 55.580 s
- Data length query: 21.457 s
- Raw data format: 35.522 s
- Data conversion formula: 39.735 s

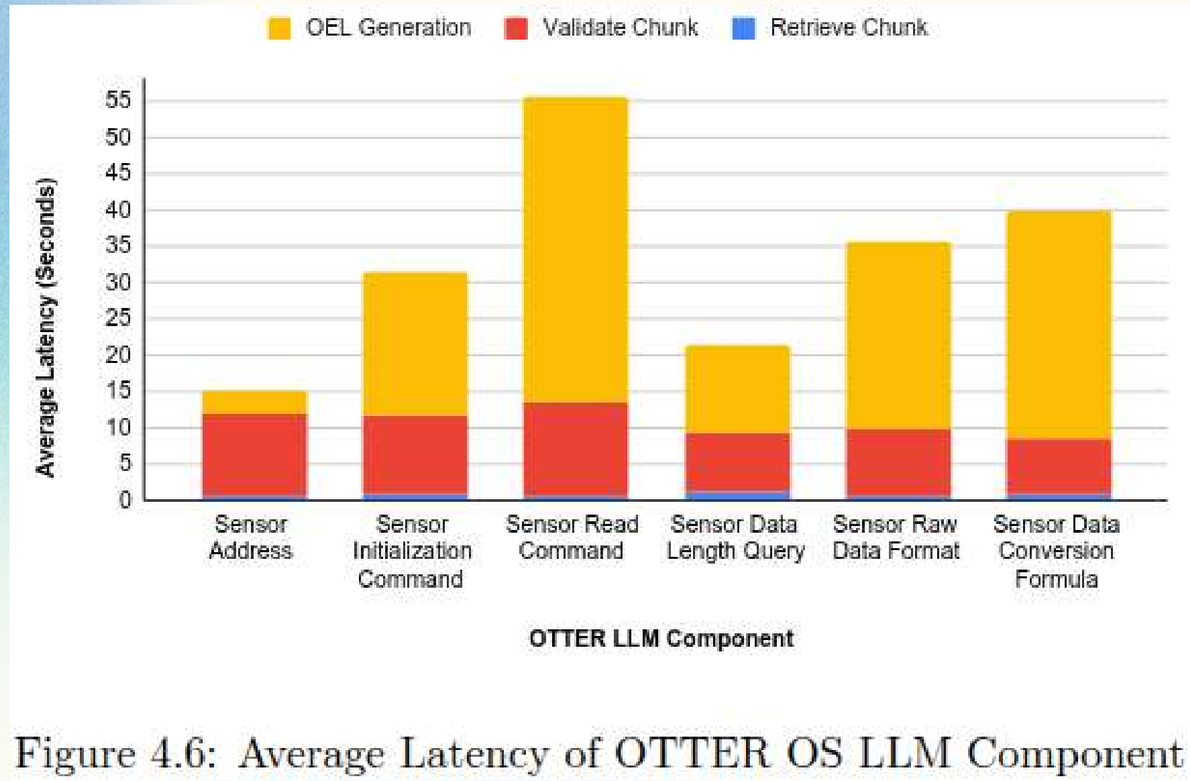


Figure 4.6: Average Latency of OTTER OS LLM Component

Latency

Evaluation of 40 trials

Datasheet Preprocessing Latency

(fetch → partition (chunking) → embed)

Mean: 13.124 s

Vector DB Embedding: 2.92 s mean

Datasheet Partition: 7.95 s mean

Datasheet Fetch: 2.26 s mean

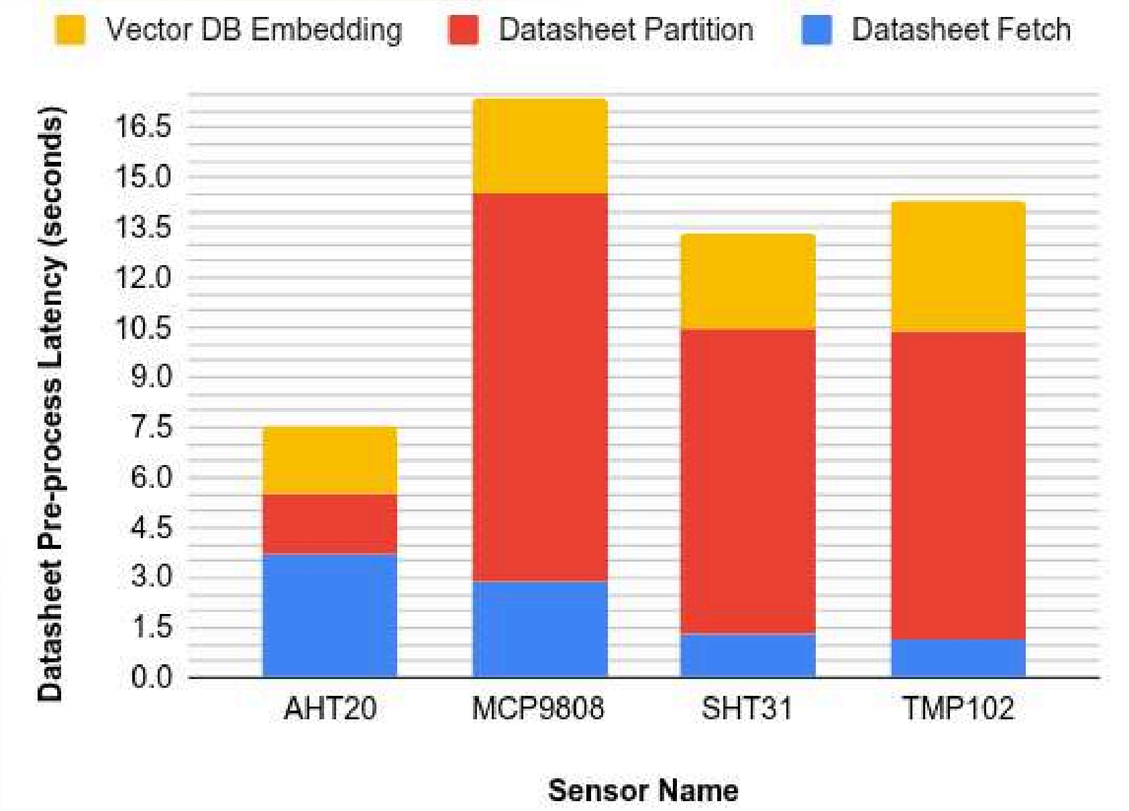
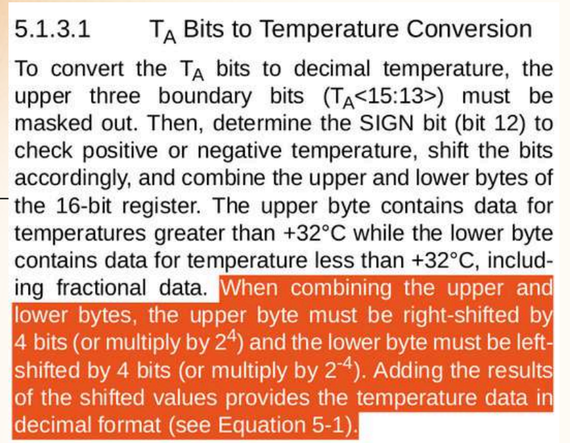


Figure 4.7: Latency of Sensor Datasheet Initial Preprocessing

OTTER LLM Intelligence, an empirical observation



5.1.3.1 T_A Bits to Temperature Conversion

To convert the T_A bits to decimal temperature, the upper three boundary bits (T_A<15:13>) must be masked out. Then, determine the SIGN bit (bit 12) to check positive or negative temperature, shift the bits accordingly, and combine the upper and lower bytes of the 16-bit register. The upper byte contains data for temperatures greater than +32°C while the lower byte contains data for temperature less than +32°C, including fractional data. When combining the upper and lower bytes, the upper byte must be right-shifted by 4 bits (or multiply by 2^4) and the lower byte must be left-shifted by 4 bits (or multiply by 2^{-4}). Adding the results of the shifted values provides the temperature data in decimal format (see Equation 5-1).

Non-Deterministic, Yet Correct

SCALE_FORMAT of MCP9808.
One answer used multiplication
One answer used division

Inferring What is Needed

RPN for TMP102, where
One used bitmasks
One used bit-shifts

Thinking Over Datasheets

OTTER LLM is not copying fixed snippets. It thinks through datasheet context and adapts it for its tools

OTTER is the first intelligent **LLM OS** to enable truly **dynamic I²C sensor interfacing** via simple natural-language prompts, allowing users to **add or remove sensors at runtime** without any manual coding or firmware recompilation.

Conclusion

Prompt-Driven Runtime Interfacing

OTTER Middleware combines a **RAG LLM**, domain-specific **OEL**, and a **multithreaded execution engine** to eliminate manual driver development, eliminate repeated firmware recompilation, and enable seamless runtime plug-prompt-and-play

Intelligent Sensor Interfacing System

OTTER OS **bridges natural-language prompts with sensor integration** through LLM. Allowing plug-prompt-and-play and hot-swap execution, **lowering the barrier** of entry for non-expert and **accelerating prototyping** and hardware iteration for developers

Demonstrated Efficacy

Achieved an **80.5 % end-to-end success** rate across four diverse sensors, **94.85 % chunk-validation** accuracy, **real-time hot-swapping**, and **concurrent** thread scheduling on a single I²C bus.

Limitations

High Latency

OEL code generation via sequential chain-of-thought and feedback prompting dominates runtime (~134 s per sensor), limiting time-sensitive applications.

Protocol & Sensor Coverage Gaps

Current support is restricted to simple I²C devices; sensors requiring multi-register reads, precise timing (e.g., VL53L0X), or non-I²C protocols remain unsupported.

Functional Accuracy Variance

Bit-range misalignments, sign-polarity errors, and scaling formula mistakes account for ~19.5 % run failures, highlighting sensitivity to datasheet inconsistencies and ambiguous register descriptions.

Future Works

Latency Optimization

Explore batched or **distilled prompts**, edge-deployed LLMs, and **parallel query execution** to reduce reliance on cloud inference.

Expanded Sensor & Protocol Support

Add **SPI/UART drivers**, multi-register sequencing, custom timing workflows, and broader device compatibility for industrial use cases.

OEL Accuracy & Enhancements

Improve accuracy against non-standard datasheets and ambiguous documentation by tweaking **embedder or parser.** Enrich OEL with sampling intervals, resolution settings, and event triggers.

Live Demo and Q&A Session

Thank you for listening!