ABSTRACT

1

INTRODUCTION

2

LITERATURE REVIEW

3

DESIGN

In this section, we present the hardware equipment and software programs used to build the OTTER OS as well as to run and evaluate the experiments. We will also showcase the system design that includes the OTTER OS architecture, OTTER embedded language, OTTER LLM, OTTER Engine, and the Embedded OS. Next, we present the design for the experiments to test the capabilities of how well the OTTER OS is able to interface with numerous I2C sensors.

3.1 Hardware Equipment Used

3.1.1 Arduino Nano 33 BLE Sense Lite

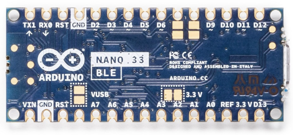


Figure 3.1: Arduino Nano 33 BLE

Arduino Nano 33 BLE Sense Lite is a tiny and cost-effective Bluetooth-enabled development board that utilizes the nRF52840 micro-controller. The board runs on Arm Mbed OS with 256KB SRAM and 1 MB Flash. However, the choice of the board is mainly because of the ubiquitousness of the Arduino platform and the hardware availability of the laboratory, as the OTTER OS is meant to be micro-controller and OS agnostic.

3.1.2 I2C Sensors

In this project, various I2C sensors are used to evaluate the runtime interfacing capabilities of OTTER OS. The Inter-Integrated Circuit (I2C) is chosen due to its simplicity, two-wire configuration, ubiquity, and support for multi-device communication, which aligns with the modular and dynamic design goals of OTTER OS.

We chose sensors with a diverse range of functionality and manufacturers. This allows us to evaluate the system across different data formats, scaling behaviors, and I2C addressing schemes. The I2C sensors used include:

- **TMP102** – A temperature sensor from Texas Instruments that provides 12-bit temperature readings and operates at 0.0625°C resolution.

- **MCP9808** – A high-accuracy temperature sensor from Microchip Technology with configurable alert output and ±0.25°C accuracy.
- **AHT20** – A digital temperature and humidity sensor from ASAIR with factory-calibrated, compensated output.
- **SHT31** – A sensor from Sensirion offering high-accuracy temperature and humidity measurement over I2C.
- **MPL3115A2** – A combined pressure and altitude sensor with an I2C interface, suitable for measuring atmospheric changes.
- **IRS11A (0J9776)** – A time-of-flight (ToF) distance sensor providing high-speed proximity measurements over I2C.

3.2 Software Libraries Used

3.2.1 MbedOS

MbedOS is an open-source real-time operating system (RTOS) developed by ARM for ARM Cortex-M micro-controllers. MBed OS provides features such as task scheduling, hardware abstraction, and peripheral drivers, which are used by OTTER OS to enable dynamic sensor interfacing at runtime.

MbedOS is chosen because it is natively supported by Arduino Nano 33 BLE Sense Lite nRF52840 micro-controller. MbedOS has all of the essential RTOS feature described previously. Mbed OS uses C++ that aligns with language used for OTTER Engine. Lastly, MbedOS is used widely amongst the embedded system community.

3.2.2 Duck Duck Go Search

Duck duck go is a search engine that is used to find the official I2C datasheet on the internet. We utilize the search engine API, and appends the search with filetype:pdf to find downloadable datasheet pdf. If a matching datasheet is found, it is downloaded and stored locally for processing. This module will allows OTTER to find data of any I2C sensor available on the internet.

3.2.3 PyMuPDF

PyMuPDF is an open-source PDF extractor. It is able to parse complex datasheet formatting including tables, bullet points, and code blocks. The parsed information are put into a markdown file to preserve the formatting and details. This extraction is necessary to do structured chunking with minimal loss.

3.2.4 LangChain

LangChain is an open-source orchestration framework for developing large language models. We use LangChain to do chunking of the datasheet markdown. This is splitting the datasheet into smaller chunks with overlap in between. Each chunks are then stored in LangChain's Document object.

3.2.5 OpenAI Ada-002 Embedding Model

Ada-002 is a closed-source embedding model developed by OpenAI. Ada-002 converts text into high-dimensional vector representations. Words with similar meanings (”cat” vs ”dog”) will have higher cosine similarity between their vector representation. While, words with different meanings (”cat” vs ”car”) will have distant cosine similarity between their vector representation, even though their character are similar. In this project, Ada-002 is used to embed the chunked datasheet content into dense vector.

3.2.6 Meta FAISS

FAISS is an open-source library by Meta for similarity search and clustering of dense vectors. FAISS can search in sets of vectors of any size, including the ones that is larger than the system memory. We use FAISS to store and index datasheet chunk vectors. We also use FAISS to find and retrieve relevant chunks based on user queries by finding the cosine similarity between the query vector and the stored datasheet vectors.

3.2.7 OpenAI GPT o3-mini

GPT o3-mini is a closed-source lightweight large language model from OpenAI. It is capable of understanding and generating natural language. In OTTER, we use o3-mini to interpret user prompts, and derive the sensor parameters from the datasheet chunks. We also use o3-mini to do chunk validation, chain-of-thought, feedback, and cleanup. We will explain each of these functionality in the system design of OTTER OS.

3.3 System Design

We first present the OTTER LLM OS Architecture as a whole system. We then present each of the component contained within the OTTER OS. Additionally, the OTTER Embedded Language (OEL) will also be discussed.

3.3.1 OTTER LLM OS Architecture

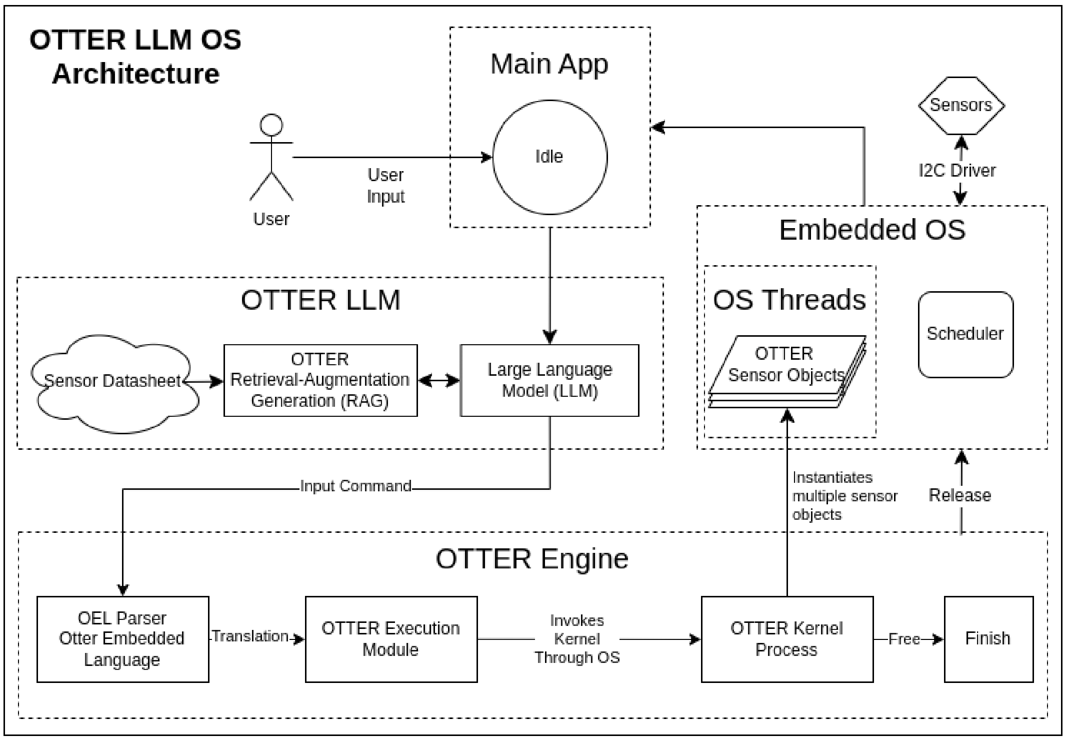


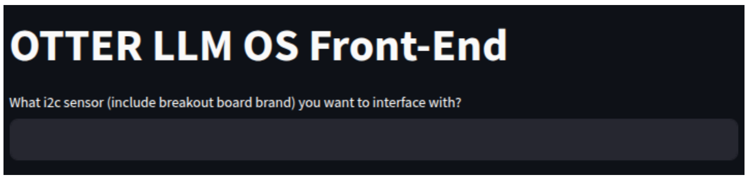
Figure 3.2: OTTER OS Architecture

Figure 3.2 presents the high-level architecture of OTTER LLM OS. The OTTER LLM OS Architecture lives one abstraction layer above the OS Abstraction Layer (OSAL), in this case the embedded OS. Contrary to traditional operating systems, OTTER focuses more on injecting intelligence into the existing operating system itself, instead of being a substitution of it. OTTER LLM OS consists of 4 major subcomponent,

- **Main App** as the front-end user interface.
- **OTTER LLM** as the intelligent interface.
- **OTTER Engine** as the relay and translation component from OTTER LLM to the Embedded OS.
- **Embedded OS** as the OS/RTOS that executes OTTER processes using multiple threads to support concurrency. Uses I2C drivers to communicate with sensors through I2C bus.

The system is designed to dynamically interface with I2C sensors at runtime, using a layered modular approach that integrates embedded software, LLM and Retrieval Augmentation Generation (RAG), and a domain-specific language (OTTER Embedded Language).

3.3.2 Main App


OTTER LLM OS Front-End

What i2c sensor (include breakout board brand) you want to interface with?

Figure 3.3: OTTER Main App Front-End

At the top level, the user interfaces with the front-end and initiates a query of sensor name to the main app as shown in 3.3. The query is then passed on to the OTTER LLM 3.3.4.

3.3.3 OTTER Embedded Language

The design of OEL follows the general pipeline of how to interface with I2C sensors. The steps can be broken down into 4 sections: Begin transmission to I2C address, initialize I2C sensor, trigger a sensor read, and process and convert incoming data into SI units. Below is how OEL looks like, where the angle brackets represent arguments to be passed in

```
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>;
```

The example usage of OEL to interface with AHT20 is as follows:

```
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 -");
```

The OEL consists of 10 variables, the NEW_SENSOR takes in the sensor name string, PROTOCOL takes in the hardware protocol as OTTER is meant to be protocol agnostic in future research, SENSOR_ADDR takes in the I2C hexadecimal address of the sensor, INIT_CMD takes in the hexadecimal register address sequence that initializes the I2C sensor, NEW_SENSOR_READ is a command that to enable read in the OTTER Engine for the particular sensor, READ_CMD takes in the hexadecimal register address to trigger I2C sensor read, DATA_LEN takes in the size of the data coming in that is requested from I2C sensor, DATA_KEY_VAL takes in a hash map where the key is an integer and the value is the expected data label, DATA_FORMAT takes in a hash map where the key is an integer and the value is the range in which the data is stored in the raw data, and SCALE_FORMAT takes in a hash map where the key is an integer and the value is a string in Reverse Polish Notation (RPN) denoting how to convert the raw data into the SI unit.

The last three variables are connected with each other. The same key has the property of all three variables. This was done this way because micro-controllers typically lack native support for hash maps.

3.3.4 OTTER LLM

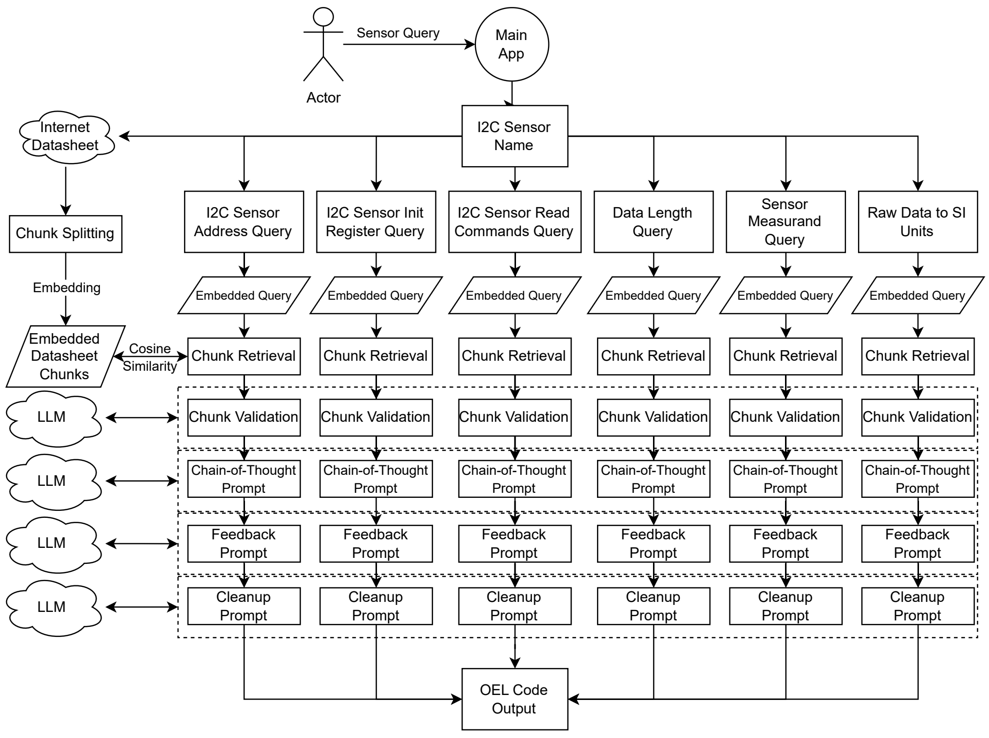


Figure 3.4: OTTER LLM Process

The OTTER LLM design can be seen from Figure 3.4. After the sensor name is received from the main app, OTTER LLM does two things: datasheet processing, and OEL code generator.

Datasheet Processing. OTTER LLM either finds the sensor datasheet on the internet or accepts file upload from the user. After retrieving the datasheet, it is then split into smaller chunks and embeds it using an embedding model. The embedded chunks are then indexed and stored into a vector database for retrieval purposes.

OEL Code Generator. The OTTER LLM processes the queries needed to generate the relevant OEL Code for the sensor. The OEL Code output is a combination of all 6 query output. For each query process, there are 7 steps.

Initialize Query. The first step is the populate the query with the sensor name.

Query Embedding. The second step is to embed the query using the same embedding that we use for datasheet chunk embedding. This returns a vector representation of the query.

Chunk Retrieval. The third step is to find the datasheet chunk that contains the query information by computing the vector cosine similarity. The cosine similarity between the query vector $\vec{q}$ and each datasheet chunk vector $\vec{d_i}$ in the vector database is computed as:

$$\text{cosine\_sim}(\vec{q}, \vec{d_i}) = \frac{\vec{q} \cdot \vec{d_i}}{\|\vec{q}\| \|\vec{d_i}\|} \tag{3.1}$$

The top 3 vectors with the highest cosine similarity scores are selected as the most relevant chunks for the given query.

Chunk Validation. The fourth step is to validate the chunk by passing in the chunk to the LLM with a prompt ”Is this `chunk` helpful for the `query`?” We found that the chunks we retrieve from step 3 alone is sometimes not useful. This is because sensor datasheets are typically written structurally for humans and in a domain-specific format, often using dense technical language, tables, and register maps that differ significantly from general natural language. Furthermore, the structure and terminology used in sensor datasheets vary widely across different manufacturers, making it challenging for generic text embeddings to consistently capture meaningful relationships. As a result, the semantic embeddings—optimized for general-purpose language—may fail to reflect the true contextual relevance of these sections in relation to a specific query. This discrepancy makes validation through LLM reasoning a critical step to filter out semantically similar but functionally irrelevant chunks.

Chain-of-Thought Prompting. The fifth step involves chain-of-thought (CoT) prompting to guide the LLM in producing accurate and interpretable output. First, a system prompt is provided: `"You are a helpful assistant and an expert in I2C sensors. Assume ideal and default conditions."`. Second, the consolidated and validated chunks from the previous step are inserted into the prompt and labeled as `context`. Third, the user's `query` is appended to specify the desired output. Fourth, explicit rules are included to constrain the response, such as disallowing conditional statements (e.g., `if`) and enforcing the use of valid operators. Finally, chain-of-thought reasoning is initiated with the prompt: `"Please explain your reasoning step by step, using both the context and your internal knowledge."` This structured prompting strategy encourages the model to produce transparent, rule-abiding, and verifiable outputs grounded in both the retrieved datasheet content and its own learned knowledge.

Feedback Prompting. Since the output from the chain-of-thought (CoT) prompt is expressed as a logical step-by-step reasoning process, it cannot be directly translated into OTTER Embedded Language (OEL). Instead, we must extract only the final answer relevant to the original query. To accomplish this, the CoT output is fed back into the LLM as part of a new prompt, labeled as: `"My expert told me:"`. This is followed by an instruction such as: `"Please provide the (query). Extract only the (query)."` By reusing the expert-generated context while focusing the LLM on extracting a concise answer, this feedback prompting technique significantly improves the accuracy and reliability of the response, making the response useful.

Cleanup Prompting. The cleanup prompting stage serves to refine and format the output for integration into the OTTER Embedded Language (OEL). While the feedback prompt typically provides a concise answer, it may still contain extraneous text or lack the precise structure required for OEL. In this step, the feedback output is passed once more to the LLM, labeled as: `"My expert told me:"`. This is followed by a formatting instruction, such as: `"Please arrange it into (arrangement)."` To further enforce structural conformity, we include an additional prompt: `"ONLY FILL IN the sentence. The (query) is arranged as: (arrangement)."` This ensures that the final output is both syntactically clean and semantically aligned with the expected OEL format.

OEL Code Output. In the final stage, the output from the six query prompts-each representing a specific aspect of the sensor configuration-are consolidated and arranged into a complete OTTER Embedded Language (OEL) code.

3.3.5 OTTER Engine

The OTTER Engine purpose is to relay and translate OEL from OTTER LLM to the Embedded OS. The OTTER Engine has 3 subcomponents: OEL Parser, OTTER Execution Module, OTTER Kernel Process.

OTTER Execution Module. The OTTER Execution Module acts as the system's interface controller and entry point for runtime sensor operations. It resides in the main loop and constantly polls the serial buffer for user-issued commands. Its primary responsibility is to bridge human-readable prompt outputs (in OEL format) and internal runtime operations by orchestrating command parsing, validation, and delegation.

When a command such as `NEW_SENSOR` is received, the Execution Module:

- Extracts and cleans the input string.
- Passes the raw OEL string to the OEL Parser to generate a SensorConfig object.
- Prints the parsed configuration fields to the serial console for user verification.
- Forwards the structured configuration to the OTTER Kernel Process to instantiate or manage the corresponding sensor thread.

For commands like `SENSOR_REMOVE`, the module retrieves the sensor name, validates its existence in the thread registry, and signals the kernel to perform teardown. By isolating all external input parsing and validation to this module, OTTER OS adheres to a clean separation of concerns—maintaining robustness and preventing malformed user commands from directly affecting system-critical execution paths.

This module also ensures responsiveness by introducing lightweight scheduling (sleep_for(1ms)) and refrains from performing any blocking or sensor-specific operations directly. It acts purely as a coordinator—never managing thread lifecycles or sensor states directly, ensuring the core execution remains encapsulated within the kernel.

OEL Parser. The OEL Parser serves as the syntactic and semantic interpreter of OTTER OS, transforming high-level, LLM-generated OEL (OTTER Embedded Language) code into a structured runtime object: the SensorConfig. This transformation is fundamental to OTTER's mission of natural language-driven sensor interfacing, as it establishes the bridge between abstract user intent and low-level, executable logic.

The OEL format is a domain-specific configuration language consisting of key-value directives, each representing a discrete semantic function in the sensor communication pipeline. These directives may include primitive scalar fields (`SENSOR_ADDR`), nested byte sequences (`INIT_CMD(0x01, 0x0A)`), or compound symbolic mappings (`DATA_KEY_VAL(0:"Temp")`).

The parser operates as a multi-stage interpreter:

- Lexical Processing: It first tokenizes the OEL string using semicolon delimiters to isolate each command. It then identifies the presence of an equals sign (`=`) to split commands into identifiers and values.

- Semantic Routing: Each directive is then dispatched to a specialized handler based on its command name. For example, commands such as `INIT_CMD` and `READ_CMD` trigger nested parsing routines to extract byte arrays, while `DATA_KEY_VAL` and `SCALE_FORMAT` require handling of both structural delimiters and symbolic content.

- Sanitization & Normalization: Strings are trimmed, quotation marks removed, and hexadecimal and decimal representations are interpreted dynamically. This ensures the parser can support OEL code generated from a wide range of LLM prompt styles and formatting variations.

One of the parser's most critical tasks is to ensure correctness and isolation between different configuration fields. It must safely and predictably extract elements such as register addresses, bit ranges, and mathematical transformation expressions while preserving execution integrity. As such, special care is taken to validate each field's structure and enforce constraints—for example, checking for matching parentheses, properly splitting colon-delimited mappings, and converting RPN strings into scalable stack-evaluable formats.

Additionally, the parser is designed with extensibility in mind. By routing each command through a dispatcher pattern, new directives can be added to the OEL language without breaking backward compatibility. This opens the door for future extensions such as `SENSOR_TYPE`, `RESOLUTION`, or `SAMPLING_RATE`, which can be appended with minimal change to the parser core.

The final output of the OEL Parser is a fully-populated SensorConfig object that encapsulates all required runtime metadata. This object serves as the atomic unit of execution for the OTTER Kernel Process, and defines the entire behavior of the sensor thread—from startup handshake to binary scaling. Without this translation layer, the LLM-generated intent would remain inert; with it, OTTER achieves full automation in transforming language into live, concurrent sensor execution.

OTTER Kernel Process. The OTTER Kernel Process is the internal runtime manager responsible for the full lifecycle of sensor execution threads. Once it receives a valid SensorConfig from the Execution Module, it performs a series of well-defined actions to bring the sensor online:

- Allocates heap memory to persist the configuration struct.
- Dynamically constructs a Thread object using Mbed OS primitives, assigning a name derived from the sensor label to support future lookup and control.
- Starts the thread by binding it to the thread_function() routine, which initiates and perpetually manages the sensor's I2C communication cycle.
- Registers the thread in a global map indexed by sensor name, enabling future control operations such as removal, suspension, or debugging.

On the removal side, the Kernel Process supports thread termination through name-based lookup. It safely calls terminate() on the thread, deallocates the thread object and associated resources, and removes the sensor entry from the global registry. This process is explicitly designed to avoid memory leaks or dangling references, which are common issues in long-running embedded systems with dynamic memory.

A key design feature of the OTTER Kernel Process is its non-blocking, asynchronous behavior. It defers all time-sensitive operations (like I2C reads and bit manipulation) to the thread context, allowing the kernel to remain free for management operations. This architecture enables scalability: the system can host multiple concurrent sensor objects without central bottlenecks. The kernel operates much like a lightweight process scheduler, enabling plug-and-play capability at runtime and minimizing coupling between the application logic and sensor hardware.

3.3.6 Embedded OS

The Embedded OS in OTTER is designed to be operating system agnostic, requiring only two core features: multithreading and timing control. In the current implementation, we leverage Arm Mbed OS due to its lightweight thread scheduler, priority management, and native mutex support. These features are essential to enable the dynamic and concurrent execution of sensor routines, each operating independently in its own thread context.

The Embedded OS is not responsible for sensor logic directly, but instead acts as the execution substrate on which all OTTER Sensor Object threads run. Threads are spawned at runtime using a dynamic thread constructor, and the OS scheduler interleaves their execution based on fixed time delays and priority settings. This allows OTTER to run multiple sensors in parallel—each with different polling intervals, sensor logic, and communication overhead—without requiring task coordination in user code.

Additionally, the use of mutual exclusion locks `i2cMutex` ensures that concurrent access to shared hardware resources such as the I2C bus is serialized, preventing data corruption or bus contention. The OS's ability to handle real-time delays and timing (via `ThisThread::sleep_for`) is also used to tune the responsiveness of each sensor, ensuring reliable data collection under different sensor response profiles. Thus, while minimal, the Embedded OS is a critical enabler of concurrency and responsiveness in OTTER's runtime architecture.

OTTER Sensor Object. Each OTTER Sensor Object is instantiated as a dedicated thread in the Embedded OS and encapsulates the full I2C interaction and data processing lifecycle for a given sensor. Upon initialization, the thread begins by sending the sensor's wake-up or configuration command sequence over the I2C bus via `init_i2c()`, using mutex locks to ensure exclusive access to the shared bus. Once the sensor is initialized, the thread enters an infinite polling loop, executing sensor reads at fixed intervals.

The core of this polling loop involves three stages: (1) issuing I2C read commands, (2) collecting raw byte responses, and (3) post-processing the binary response into human-readable values. The I2C communication is handled using a blocking read strategy with timeouts, ensuring that the thread remains deterministic even if a sensor is unresponsive. Once a valid response is received, the thread invokes `process_data()` to parse meaningful fields from the byte stream. This involves extracting bitfields using programmable start and end bits defined in the OEL, followed by evaluating a reverse Polish notation (RPN) expression via `applyRPNScaling()` to convert the raw value into real-world units.

The use of dedicated threads for each sensor enables parallelism without explicit coordination between sensors, as each thread operates autonomously. This design also isolates sensor-specific failures—an error in one thread does not crash or block the execution of others. By combining thread-local logic with shared runtime services (mutexes, delay control, and memory), the OTTER Sensor Object model achieves scalability, modularity, and runtime adaptability essential for plug-and-play embedded sensing.

3.4 Experimental Design

Sensor Compatibility Experiment. The purpose of this experiment is to evaluate the hypothesis that OTTER OS can successfully support and interface with a diverse range of I2C sensors. The selected sensor set includes: TMP102, MCP9808, AHT20, SHT31, MPL3115A2, and IRS11A (0J9776), each chosen to represent a spectrum of complexity based on datasheet length, protocol quirks, and register configurations. This experiment is decomposed into multiple validation stages namely, datasheet fetching, RAG chunk, OEL Code LLM Generation, and I2C output.

Datasheet Fetching. In this sub-experiment, we test whether OTTER LLM can successfully retrieve the correct datasheet PDF given only a sensor name. The system issues a web query using DuckDuckGo with the search pattern "`<sensor_name> datasheet filetype:pdf`", and attempts to download and store the result locally. We define success as retrieving a working, complete, and manufacturer-authentic datasheet. We log the sensor name, URL, response status, and any fallback behavior in case the first result is not usable. This step validates the effectiveness of OTTER's online sensor information acquisition strategy.

RAG Chunk. After a datasheet is fetched, the document is parsed into structured markdown and chunked using LangChain. This sub-experiment measures the accuracy of the retrieval-augmented generation (RAG) pipeline by evaluating whether the retrieved chunks are contextually relevant to the prompt. We use an internal validation prompt—*"Is this chunk helpful for the query?"*—and manually annotate true relevance. This stage tests whether vector embeddings and chunk retrieval using FAISS succeed in narrowing down to semantically meaningful technical content, despite datasheets having non-standard formatting. Moreover, we will also test out whether the LLM validation is able to segregate helpful from unhelpful chunk. The ground truth for this validation is manual comparison with the datasheet.

OEL Code LLM Generation. Here, we evaluate whether the LLM can produce correct OEL code given a user prompt and a set of retrieved datasheet chunks. We test the LLM across six distinct sub-queries (sensor address, initialization command, read command, expected data length, data bit formatting, and scaling logic) and compare each output to a ground truth specification crafted from the sensor's datasheet. The correctness is scored at the field level to understand strengths and weaknesses in semantic interpretation and reasoning. This experiment serves as a measure of both generation quality and alignment to datasheet semantics.

OTTER OS I2C Output. This stage tests the complete runtime functionality of OTTER OS using the generated OEL script. The script is parsed into a SensorConfig struct and instantiated as a sensor thread within the system. We consider the test successful if the thread can read data via I2C and the final output is numerically correct and meaningful (e.g., non-zero, within physical bounds). This experiment confirms end-to-end operability, integrating the LLM, parser, and embedded runtime layers.

Multiple Sensor Experiment. The goal of this experiment is to evaluate whether OTTER OS can interface with multiple I2C sensors simultaneously. We incrementally increase the number of connected sensors and prompt OTTER OS to configure and spawn sensor threads for each of them. We monitor whether each sensor operates correctly in parallel—producing real-time output without bus collisions or timing issues. This experiment validates the multi-threaded design and runtime scalability of the system, particularly the thread management and I2C bus arbitration.

Hot-Swapping Plug-and-Play Experiment. This experiment tests the dynamic behavior of OTTER OS in response to sensor addition and removal at runtime. We begin by initializing a set of sensors and then introduce a new sensor during execution using the NEW_SENSOR command. Later, we remove an active sensor thread using `SENSOR_REMOVE`. We track whether the system can correctly allocate, spawn, terminate, and deallocate sensor threads without requiring a system reset or causing instability. This experiment demonstrates OTTER OS's plug-and-play capability and validates its robustness to hardware changes.

Latency Experiment. In this experiment, we evaluate the latency of OTTER OS from prompt to usable I2C output across sensors of varying complexity. We break down latency into components: datasheet fetching, chunk embedding, LLM reasoning, OEL parsing, and sensor initialization time. Each sensor is tested three times and averaged to ensure consistency. This helps determine whether the system remains responsive and predictable, especially when dealing with sensors with long initialization commands or complex scaling logic.

Robustness to Prompt Variations. This final experiment evaluates the ability of OT-

TER LLM to generalize over diverse prompt phrasing. For each sensor, we test multiple natural language variations (e.g., `"Get temp from TMP102"`, `"Measure TMP102"`, `"Read temperature from Texas Instruments TMP102"`) and introduce minor typos or casing errors. We compare the final generated OEL code to the canonical one to evaluate semantic robustness. This experiment validates the system's resilience to prompt variability—critical for usability by non-technical users.

3.4.1 Experiment Setup

The experimental setup consists of an Arduino Nano 33 BLE Sense Lite microcontroller running OTTER OS. The microcontroller's SDA and SCL lines are connected to a shared I2C bus via a breadboard, allowing multiple sensors to communicate over the same bus. Each I2C sensor under test is wired to this bus, powered by the Arduino's 3.3V output, and grounded to a common ground rail to ensure signal integrity. The Arduino is connected to a laptop via USB, serving both as a power supply and as a serial interface for command input and real-time output monitoring. On the host laptop, the OTTER LLM software stack is executed in a Linux environment. The laptop maintains an internet connection to facilitate external API calls for embedding generation and large language model (LLM) inference through OpenAI services.

4

EVALUATION

In this chapter, we present the analysis and results from the experiments that we conducted. Firstly, we show that the OTTER OS is 80.5% successful in interfacing with I2C Sensors. Secondly, we show that the OTTER LLM context validation is 95% accurate in identifying helpful and not helpful context. Lastly, we present the OTTER OS average latency to be 202.035 seconds.

4.1 Sensor Compatibility Experiment

The sensor compatibility evaluates the OTTER Embedded Language (OEL) and OTTER Engine. This stage tests the complete runtime functionality of OTTER OS using the generated OEL script. The script is parsed into a SensorConfig struct and instantiated as a sensor thread within the system. We consider the test successful if the thread can read data via I2C and the final output is numerically correct and meaningful (e.g., non-zero, within physical bounds). This experiment did not use LLM to produce the sample code, rather the author manually interface with the I2C sensor. Out of the 6 sensors used, we are able to write OEL code that runs the OTTER Engine correctly on 4 sensors. The sensors are AHT20, MCP9808, SHT31, and TMP102. While the sensors that are not able to produce a correct output are MPL3115A2 and VL530X. This behavior is observed because MPL3115A2 splits its reading across multiple registers, therefore we can not continuously read N bytes. Moreover, the VL530X time-of-flight sensor requires timing synchronization, and the datasheet did not provide adequate information to interface with the I2C sensor manually-instead the datasheet suggested the usage of VL530X API. Because of this result, only the four working sensors for our further experiments.

4.2 OTTER OS End-to-End Evaluation

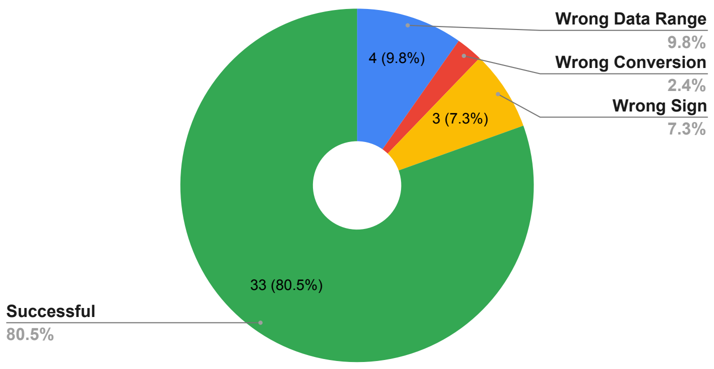


Figure 4.1: OTTER OS Accuracy

This experiment evaluates the end-to-end functional accuracy of the OTTER OS pipeline, from prompt-based sensor configuration to real-time I2C data acquisition and output. A total of 40 evaluation runs were conducted, with 10 runs performed for each of four I2C sensors: TMP102, MCP9808, AHT20, and SHT31. Each run involves issuing a natural language prompt, retrieving the sensor's datasheet, generating the corresponding OEL code, parsing the configuration, instantiating a sensor thread, and verifying the correctness of the sensor output.

Across all runs, OTTER OS achieved an overall accuracy of 80.5%. A run was considered successful if the final output value matched the expected physical reading within the correct unit, format, and sign, based on a manually verified ground truth. Among the 40 runs, six were identified as incorrect in various ways. Four of these failures were due to incorrect data range field definitions, where the bit slicing specified in the OEL did not align with the datasheet (e.g., `Temperature: [15:4]` instead of the correct `Temperature: [11:0]`). Three runs produced outputs with incorrect sign polarity, such as 25.0C when the expected result was 25.0C. Finally, one run applied an incorrect mathematical transformation when converting the raw sensor data to its physical unit, resulting in a miscalculated scaled value.

These results demonstrate the effectiveness of OTTER OS in automating sensor interfacing via LLMs and runtime execution, while also highlighting areas—such as bit-level precision and signed arithmetic—where further improvement is required.

4.3 RAG Evaluation

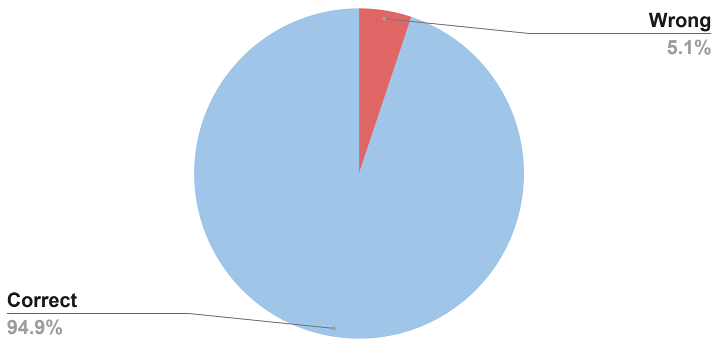


Figure 4.2: Chunk Validation Accuracy

This experiment evaluates the performance of the Retrieval-Augmented Generation (RAG) component within the OTTER LLM pipeline. Specifically, it assesses the accuracy of retrieving relevant and informative chunks from sensor datasheets to support subsequent OEL code generation. Accuracy is defined as the system's ability to distinguish and select chunks that contain information genuinely helpful for answering the user's prompt.

We measure two stages in the RAG process: (1) the initial chunk retrieval using FAISS from the vector database, and (2) the subsequent chunk validation using an LLM-based prompt that asks, *"Is this chunk helpful for the query?"*. The chunk validation step achieved an accuracy of 94.85% in correctly classifying helpful versus unhelpful chunks. This suggests that the LLM can effectively filter and reason over the retrieved content. In contrast, the FAISS-based retrieval step alone achieved 75% accuracy in selecting relevant chunks from the embedded vector space, indicating that some initial retrievals lacked the contextual specificity required for accurate sensor interpretation.

These findings highlight the complementary strengths of hybrid retrieval-generation workflows, where dense vector similarity provides a coarse filtering mechanism, and the LLM adds semantic precision during the validation stage.

4.4 Multiple Sensor and Hot-Swapping Plug-and-Play Experiment

This experiment evaluates OTTER OS's capability to support concurrent multi-sensor interfacing as well as dynamic sensor addition and removal at runtime. The system was tested with four I2C sensors connected simultaneously to a single I2C bus. OTTER OS successfully instantiated and maintained concurrent sensor threads for all four devices, with each thread executing its polling loop independently and producing continuous, accurate output. This validates the system's thread-based architecture and its ability to manage multiple I2C devices concurrently without conflict or data loss.

The experiment also tested OTTER's plug-and-play functionality through the use of the `NEW_SENSOR` and `SENSOR_REMOVE` commands. Sensors were added and removed dynamically during runtime, without requiring a system reboot. The system correctly handled thread creation, memory allocation, and I2C initialization for new sensors, as well as safe termination and deallocation for removed sensors. These results demonstrate that OTTER OS supports true hot-swapping behavior, enabling seamless runtime adaptation to changes in hardware configuration—a critical feature for scalable and user-friendly embedded sensing systems.

4.5 Latency Experiment

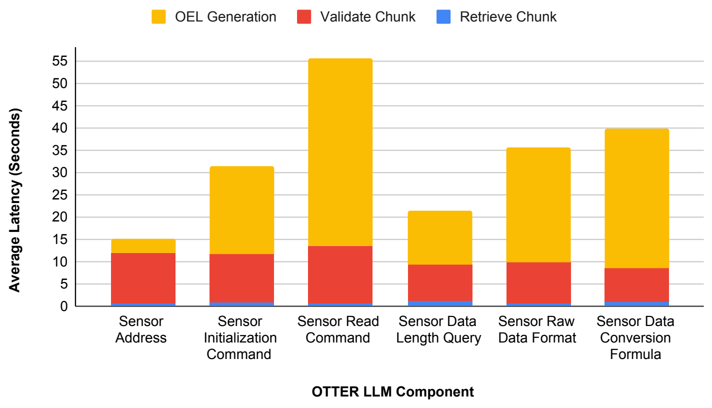


Figure 4.3: Average Latency of OTTER OS LLM Component

This experiment measures the end-to-end system latency of OTTER OS and provides a detailed breakdown of the latency across its key components. Across all tested sensors, the system demonstrated a median end-to-end latency of 202.035 seconds, with a mean latency of 200.090 seconds. The latency distribution shows an upper quartile latency of 216.731 seconds and a lower quartile latency of 182.131 seconds, reflecting some variability based on sensor complexity and response time from external APIs.

Within the OTTER LLM pipeline, individual components were profiled separately. The chunk retrieval process—responsible for selecting the most relevant datasheet sections using vector similarity—had a mean latency of 0.786 seconds. The subsequent chunk validation step, in which the LLM filters for useful content, showed a higher mean latency of 10.067 seconds due to semantic reasoning requirements. The most time-consuming stage was OEL code generation, with a mean latency of 133.869 seconds, as it involved multiple LLM invocations for different sensor configuration parameters.

We further decompose the OEL code generation into six distinct sub-queries. Among these, the sensor address query exhibited the lowest mean latency at 15.230 seconds. The initialization command component required an average of 31.464 seconds, while the read command component exhibited the highest latency at 55.580 seconds. The data length query had a mean latency of 21.457 seconds, followed by the raw data format component at 35.522 seconds, and the data conversion formula component at 39.735 seconds.

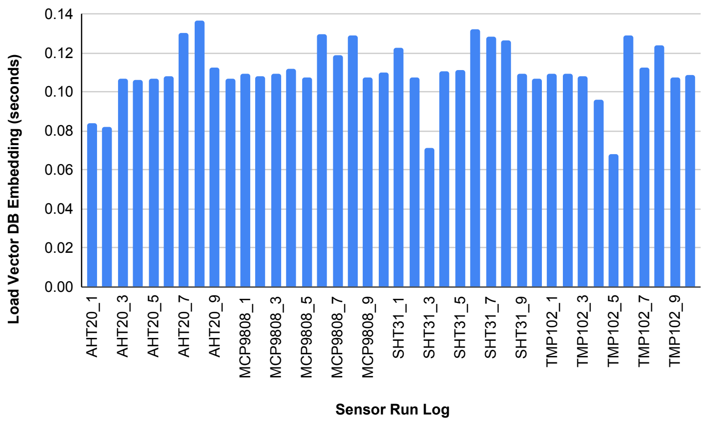


Figure 4.4: Load Vector DB Embeddings from local file

In addition to the LLM pipeline, we also measured the latency of the datasheet preprocessing phase. This includes downloading the datasheet, converting it into markdown format, splitting it into chunks, and embedding the chunks into the vector store. This phase exhibited a mean latency of 13.124 seconds.

These results suggest that while the chunk retrieval and validation stages are relatively lightweight, the LLM-driven generation of structured sensor configurations is the dominant contributor to overall system latency. Optimization of this stage or parallelization of component queries presents a potential avenue for improving system responsiveness.

5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This thesis introduced OTTER OS, a novel operating system and runtime engine designed to enable prompt-driven, dynamic I2C sensor interfacing using large language models. OTTER OS bridges the gap between natural language instructions and low-level embedded sensor operations by combining a layered software-hardware architecture with intelligent reasoning, dynamic thread management, and runtime configurability. At its core, the system leverages a retrieval-augmented generation (RAG) pipeline, a domain-specific language (OTTER Embedded Language), and a modular execution engine running on a multithreaded Embedded OS.

Through detailed system design and experimentation, we demonstrated that OTTER OS can support seamless integration with multiple I2C sensors—such as TMP102, MCP9808, AHT20, and SHT31—achieving a full end-to-end success rate of 80.5%. The system supports real-time sensor execution, hot-swapping via prompt commands, and concurrent thread scheduling on a shared I2C bus. The retrieval and chunk validation process within the OTTER LLM achieved a high accuracy of 94.85%, confirming the LLM's ability to filter and reason over unstructured datasheet content. Additionally, our latency experiments showed that most time is spent in OEL code generation, which highlights an optimization target for future development.

Importantly, the system design demonstrates several unique strengths: (1) the ability to interface with a wide range of sensors without writing or compiling code, (2) support for real-time dynamic addition and removal of sensors, and (3) an interpretable, language-level representation (OEL) that serves as the bridge between AI-generated logic and embedded execution.

5.2 Future Research Directions

Several challenges remain and motivate future work. First, latency remains a bottleneck for time-sensitive applications. While chunk retrieval is near-instantaneous, the chain-of-thought reasoning and feedback prompting introduce significant delay, especially when executed sequentially. Future work should explore batch LLM prompting, prompt distillation, or edge-deployable models to reduce dependency on cloud-based inference. Second, sensor compatibility could be improved by extending support to sensors that require multi-register reading, custom timing, or SPI/UART protocols. Adding these would make OTTER OS more versatile in complex or industrial deployments.

Third, while OEL has proven effective as a lightweight and expressive intermediate representation, further refinement—such as introducing nested object support, auto-complete suggestions, or validation schemas—could improve its scalability and developer usability. Additionally, improving LLM robustness against datasheet inconsistencies or formatting errors will be critical for generalizing beyond typical use cases.

In summary, OTTER OS demonstrates the feasibility and practicality of natural language-driven sensor programming. It showcases a shift from traditional firmware-centric workflows to dynamic, intelligent runtime systems, where sensor drivers can be configured and executed using only human-readable commands. As embedded systems become increasingly complex and personalized, this approach opens up new possibilities for adaptive, user-friendly, and intelligent edge computing.

We believe the above future research directions will advance the technology presented in this thesis and contribute to academia and industry.