OTTER OS: DYNAMIC SENSOR INTERFACING IN EMBEDDED SYSTEM LEVERAGING LARGE LANGUAGE MODELS (LLMS)

Submitted by
Steven Antya Orvala Waskito

Department of Electrical and Computer Engineering

In partial fulfillment of the
requirements for the Degree of
Bachelor of Engineering (Computer Engineering)
National University of Singapore

B.Eng Dissertation

OTTER OS: DYNAMIC SENSOR INTERFACING IN EMBEDDED SYSTEM LEVERAGING LARGE LANGUAGE MODELS (LLMS)

By
Steven Antya Orvala Waskito
National University of Singapore
2024/2025

Project ID: H325310
Project Supervisor: Prof Ambuj Varshney

Deliverables:
Report: 1 Volume
Program: 1 Github Repository `https://github.com/stevenantya/LLM-OS-OTTER`

ABSTRACT

Sensor interfacing in embedded systems traditionally requires manual driver development, firmware recompilation, and is limited to static configurations, making dynamic sensor interfacing a complex and inflexible process. This paper introduces OTTER OS, a novel embedded operating system that leverages Large Language Models (LLMs) and retrieval-augmented generation (RAG) to enable prompt-driven, runtime i^2c sensor interface. OTTER OS integrates an intelligent LLM pipeline for datasheet parsing, a domain-specific intermediate language (OTTER Embedded Language or OEL), and a multithreaded execution engine (OTTER Engine) built on Mbed OS to dynamically instantiate sensor threads from natural language commands. The system supports plug-and-play functionality, multiple concurrent sensors, and reading of i^2c data in physical measurements, all without requiring manual code updates or recompilation. Experimental results show that OTTER OS achieves an 80.5% success rate in end-to-end sensor interfacing and 94.9% accuracy in context validation. OTTER OS's modular architecture and runtime adaptability demonstrate a new paradigm in embedded sensor interfacing—bridging AI-driven reasoning with low-level hardware control to reduce development time, improve flexibility, and democratize sensor access for non-experts.

ACKNOWLEDGMENTS

I would like to thank Prof. Ambuj Varshney for his continuous support and idea contributions.

CONTENTS

1

INTRODUCTION

1.1 Sensor Interfacing

Sensors are an integral part of modern computing system[1, 2]. They allow computer to sense the environment through the measurement of physical, chemical, or biological phenomena, such as temperature, pressure, motion, light, and humidity[1, 3, 4, 2]. They work by converting the physical phenomena into electrical signals, which can then be processed, interpreted, and acted upon by embedded systems, microcontrollers, or general-purpose computers[3]. By leveraging sensors, researchers and engineers manage to create a plethora of embedded system use-cases across domains, ranging from environmental monitoring and smart agriculture to biomedical wearables and industrial automation[5, 6, 7, 8]. Additionally, as we explore more digitization of physical environment, more variations of sensors are created to address these new use cases. As a matter of fact, we now have more than 100,000 sensors in the market[9].

In traditional embedded systems, we have multiple ways to interface with a sensor. First, we can use the hardware protocol interface and program it directly from there. Developers must consult the sensor's datasheet, configure the appropriate hardware communication protocol (such as i^2c, SPI, or UART), write device-specific driver code to handle register access and data conversion, and finally recompile and flash the firmware onto the microcontroller[10, 11, 12]. Another approach to do sensor interfacing is by using sensor drivers or libraries provided by open-source contributors or companies like Adafruit and Pololu. This is much more simpler and streamlined as we can directly utilize the APIs to facilitate sensor communication[13, 14].

1.2 Motivation

However, the current sensor interfacing solutions have several problems. First, programming using hardware protocol interface programming is a complex workflow that often requires significant engineering effort[15]. Second, open-source sensor drivers or libraries only covers a subset of the available sensor in the market[13, 14, 16]. Additionally, some sensor drivers, especially for newer or specialized sensors, are proprietary, limiting accessibility[16]. Third, existing sensor interfacing solutions are statically implemented and fixed on compile time[17, 18, 19, 20]. Hence, they do not support plug-and-play or hot-swapping functionality. When a new sensor is introduced or an existing one is removed, developers must manually write new code and reprogram the microcontroller, reducing system flexibility. Fourth, while there

are many online examples for interfacing with a single sensor, there are few resources that demonstrate how to interface with multiple sensors—especially in configurations that match a user's specific hardware arrangement[17, 18, 19, 20].

There is a strong rationale for addressing these four issues. If resolved, they would enable the development of a system with a simplified workflow that can interface with virtually any sensor on the market. Such a system would support dynamic addition and removal of sensors, allow users to easily experiment with different sensors, and enable simultaneous interaction with multiple sensors. The benefits are substantial: faster prototyping, easier upgrades to existing systems, a lower barrier to entry for sensor interfacing, and the ability to leverage newly released or specialized sensors without waiting for official library support.

Therefore, there is a need for us to rethink how we do sensor interfacing.

In recent years, there has been a revolutionary shift in the capabilities of large language models (LLMs)[21]. LLMs are advanced neural networks trained on vast amounts of text data, capable of understanding and generating human-like language[22]. In addition to being conversation-companion, these models can reason over technical documentation, infer intent from natural language, and generate code across diverse programming languages[21]. Furthermore, the rise of retrieval-augmented generation (RAG) systems enables LLMs to leverage external sources—such as device datasheets and documentation—to improve their contextual accuracy without the need for retraining[23, 24]. Research has shown that the combination of LLM and RAG has proven useful in doing document QnA and knowledge retrieval from unstructured technical content[25]. By using LLM and RAG as an intelligent interface, we may be able to develop an embedded operating system that can autonomously interpret sensor datasheets, generate appropriate driver logic, and configure sensors at runtime—all from a simple user prompt. This would eliminate the need for manual driver development, reduce dependency on pre-existing libraries, and enable true plug-and-play sensor integration.

Such a system could revolutionize how embedded devices interact with hardware, transforming sensor interfacing into a dynamic, prompt-driven experience.

1.3 Problem Statement

This thesis aims to respond to the existing challenge in embedded system sensor interfacing by answering the question: *Is it possible to develop an intelligent sensor interfacing system that allows users to add and remove i^2c sensors through simple prompts, without manual coding or firmware recompilation?*

To respond to this challenge, we propose OTTER OS, a novel embedded operating system that combines a large language model (LLM), a retrieval-augmented datasheet processing pipeline, and a runtime execution engine to enable prompt-driven sensor interfacing. The system introduces a domain-specific language called OTTER Embedded Language (OEL), which captures sensor configuration in a structured and machine-executable form.

2

LITERATURE REVIEW

There has been a research effort in addressing the limitations of static sensor interfacing in embedded systems. Various papers and open-source codebases have explored different methods for enabling a more dynamic, runtime sensor integration in embedded systems [26, 27]. These efforts aim to reduce the manual overhead involved in sensor configuration, streamline the integration processes, and improve the flexibility of embedded sensors [28].

At the same time, the rapid advancement of Large Language Models (LLMs) has introduced new possibilities such as interpreting sensor datasheet, generating sensor drivers, and generating sensor interfacing code generation[29, 30, 31, 32]. On top of that, LLM multimodality and the addition of Retrieval Augmentation Generation (RAG) pipeline have allowed LLMs to access technical papers and contextualize their outputs more effectively, making them useful tools for tasks traditionally reserved for engineers [23, 24, 25].

2.1 Dynamic Sensor Interfacing

Embedded sensor platforms historically rely on static configuration and manual coding for hardware integration. Early embedded operating systems like TinyOS and Contiki examplify the traditional approach where sensor drivers and configuration are compiled into the firmware image, and changing the hardware typically meant rebuilding and redeploying the software[33, 34]. TinyOS introduced a component-based architecture with the NesC language in an effort to make sensor software modular. However, TinyOS produces a monolithic library where all components and sensor drivers are fixed at compile-time, and does not allow runtime updates[33]. Contiki and SOS then introduced dynamic loading of modules on sensor nodes, allowing partial update to the system[34, 35]. Nonetheless, these dynamic OS had limitations: for example, Contiki restricts the dynamic reconfiguration to application-level code and does not support inserting low-level drivers at runtime[34]. Similarly, Zephyr RTOS's device tree supports dynamic peripheral binding, though it still require prior compilation and knowledge of target sensor configuration[36]. In practice the "dynamic sensors interface" in embedded OS require careful pre-planning of what can be updated, and adding support for a completely new sensor post-deployment remains non-trivial. Sensor driver code must be present at the start in the firmware as a compiled module, which still assumes an expert prepared the code in advance. Traditional embedded system OS thus lacks a true dynamic sensor interfacing pipeline.

Aside from embedded system OS, high-level scripting environments like MicroPython have

emerged to ease the burden of sensor interfacing by allowing developers to write and run code directly on microcontrollers without recompilation [37]. MicroPython provides hardware-agnostic APIs and supports interactive development, enabling users to initialize i^{2}c buses and communicate with sensors using just a few lines of Python code[38]. This model accelerates prototyping and reduces the barrier to entry compared to traditional C/C++ development[27]. However, while scripting simplifies the programming interface, it still requires the developer to understand the sensor's datasheet, manually write register-level logic, and embed the necessary configuration steps in code [37, 39]. The scripts also lack modularity at runtime—reloading a new sensor driver typically requires halting the system and re-uploading the script.

In summary, while the embedded systems community has explored modular drivers, dynamic loading, and script-based flexibility, most solutions still fall short of enabling runtime sensor integration that is fully dynamic, declarative, and responsive to new or unknown hardware. A truly dynamic sensor interface would require the ability to ingest new sensor specifications on-the-fly, generate the necessary configuration logic at runtime, and manage the execution without interrupting the rest of the system—a capability not yet fully realized in traditional embedded system designs.

2.2 Large Language Models in Embedded System

For decades, researchers and developers have sought to simplify sensor interfacing by introducing libraries, hardware abstraction layers (HALs), and component-specific drivers [40]. One early example was the release of `i2c-hal`, an embedded abstraction library that attempted to eliminate the need to read datasheets or manually implement register-level i^{2}c communication [41]. With its promise of "saying goodbye to reading device datasheets or learning complex i^{2}c interactions," `i2c-hal` offered high-level abstractions for popular sensors. However, the system only supported a limited number of devices—eight i^{2}c sensors at the time of its release—and required all supported sensors to be explicitly defined and compiled into the firmware [41]. As such, the goal of fully abstracting sensor complexity was not achieved, particularly for edge cases, newly released components, or custom hardware setups.

With technological advancements, the landscape has shifted dramatically. The rise of no-code platforms, AI-driven developer tools, and large language models (LLMs) has revolutionized embedded software development [42, 43]. Tools such as GitHub Copilot, Cursor, and Replit Ghostwriter now enable developers to generate sensor interface code by simply describing their intent in natural language [32, 44, 45]. For instance, a user may input a prompt like "Read temperature from MCP9808 via i^{2}c using Arduino," and receive a func-

tional code snippet complete with initialization, register access, and data conversion logic. These tools are especially powerful because LLMs can reason over technical documentation, infer required register operations, and generate idiomatic code for various embedded platforms [29].

In the academic sphere, LLMs have been explored for tasks such as synthesizing sensor drivers, parsing device datasheets, and even debugging runtime behavior [46]. Researchers have shown that LLMs can interpret natural language instructions and generate low-level hardware configuration code, enabling more intuitive interaction with embedded systems without manual driver implementation [29]. LLMs can also be combined with techniques such as Retrieval-Augmented Generation (RAG), which enables them to access external context beyond their training data [23, 25]. This allows the model to retrieve relevant information—such as datasheets or documentation—on the fly, and generate more accurate, context-aware outputs.

In the startup sphere, one notable effort is showcased by embedd.it, a startup that uses generative AI and multimodal LLMs to convert sensor datasheets into structured register maps, which are then deterministically compiled into driver templates for HAL, CMSIS, or Zephyr OS [31]. Embedd.it significantly reduces the sensor driver development process, and allows developers to use the generated drivers for sensor interfacing.

However, despite these advances in code generation, most LLM-assisted solutions are still tightly coupled to the traditional compile-flash-execute workflow. That is, even if the sensor code is generated in seconds, it must still be manually compiled and flashed to the microcontroller. This introduces friction in workflows that involve testing multiple sensors, reconfiguring hardware setups, or iterating on sensor logic. Furthermore, generated code is static: once flashed, the firmware does not accommodate runtime changes. If a new sensor is introduced, an existing sensor is removed, or the configuration changes, the developer must regenerate the code, recompile, and flash the device again.

As such, while LLMs have significantly lowered the barrier to writing embedded code, they have not yet addressed the deeper challenge of dynamic runtime sensor interfacing. A future system must go beyond static code generation—it must allow LLMs to operate as real-time reasoning engines capable of parsing sensor specifications, generating configuration logic, and updating embedded systems on-the-fly without recompilation. This vision motivates the architecture and goals of OTTER OS, which seeks to tightly integrate LLM reasoning and runtime execution within a flexible embedded operating system.

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 i^{2}c sensors.

3.1 Hardware Equipment Used

3.1.1 Arduino Nano 33 BLE Sense Lite

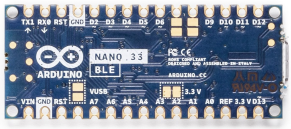

Figure 3.1: Arduino Nano 33 BLE

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 i^{2}c Sensors

In this project, various i^{2}c sensors are used to evaluate the runtime interfacing capabilities of OTTER OS. The Inter-Integrated Circuit (i^{2}c) 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 i^{2}c addressing schemes. The i^{2}c 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 i^2c.
- **MPL3115A2**: A combined pressure and altitude sensor with an i^2c interface, suitable for measuring atmospheric changes.
- **VL530X**: A time-of-flight (ToF) distance sensor providing high-speed proximity measurements over i^2c.

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 i^2c 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 i^2c 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

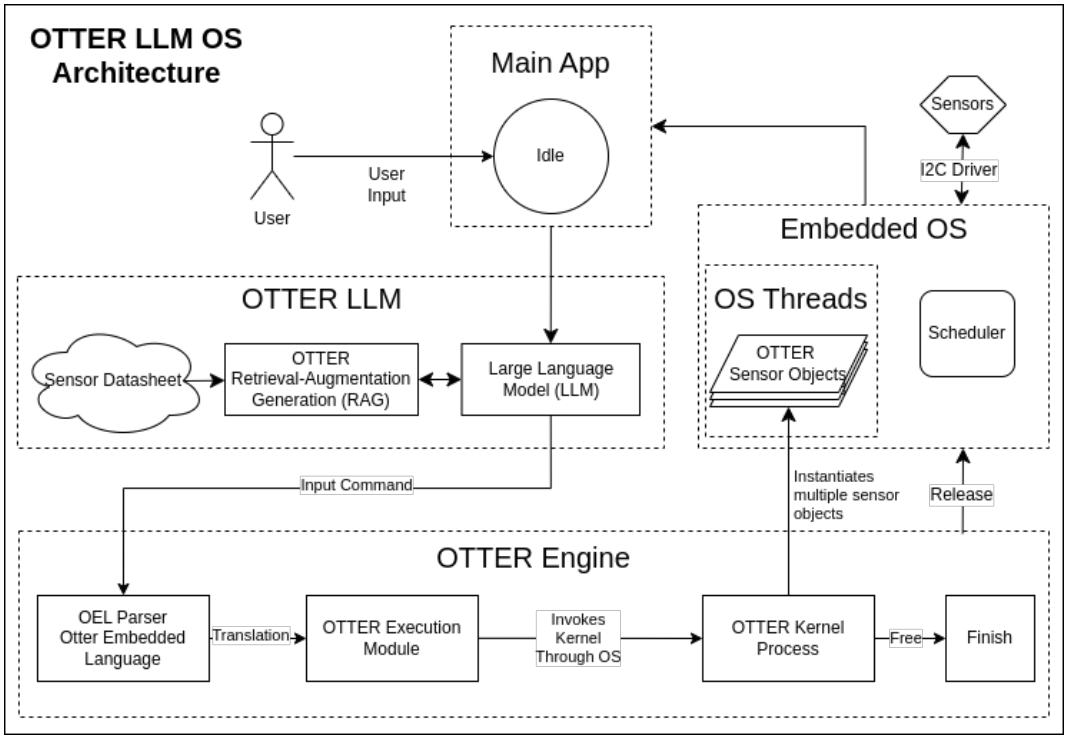

Figure 3.2: OTTER OS Architecture

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 i^2c drivers to communicate with sensors through i^2c bus.

The system is designed to dynamically interface with i^2c 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

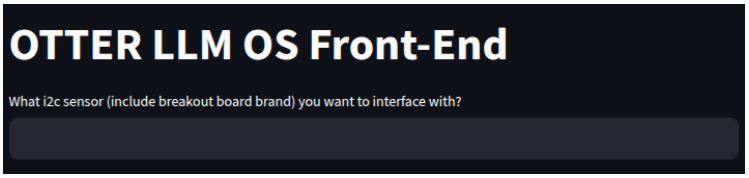

Figure 3.3: OTTER Main App Front-End

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 i^2c sensors. The steps can be broken down into 4 sections: Begin transmission to i^2c address, initialize i^2c sensor, trigger a sensor read, and process and convert incoming data into physical 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 i^2c hexadecimal address of the sensor, INIT_CMD takes in the hexadecimal register address sequence that initializes the i^2c 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 i^2c sensor read, DATA_LEN takes in the size of the data coming in that is requested from i^2c 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

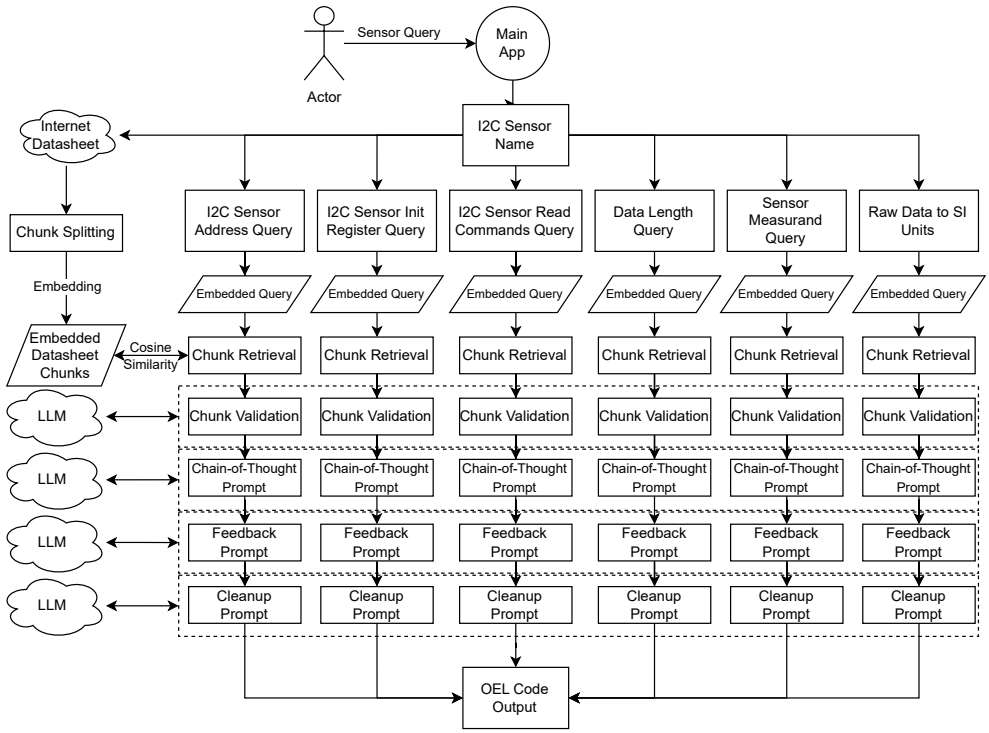

Figure 3.4: OTTER LLM Process

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 i^2c 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 i^2c 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 `i²c Mutex` 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.

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

In this experiment, we evaluate the hypothesis that OTTER OS can successfully support and interface with a diverse range of i^2c sensors. The selected sensor set includes: TMP102, MCP9808, AHT20, SHT31, MPL3115A2, and VL530X, each chosen to represent a spectrum of complexity based on datasheet length, protocol quirks, and register configurations. This experiment did not use LLM to produce the sample code, rather the author manually created the OEL Code according to the specifications on the datasheet. We consider the sensor interface is successful the OTTER OS shows the expected output data that is numerically correct and meaningful.

OTTER OS End-to-End Experiment. In this experiment, we test whether OTTER OS can interface with sensors using only the sensor name prompt as input. The evaluation is conducted accross the full OTTER OS pipeline, from user query to OTTER LLM prompt parsing and datasheet retrieval, to OEL code generation, to OTTER Engine runtime, OS thread instantiation, and lastly the real-time i^2c sensor output. The test is designed to simulate the typical interaction flow of an end-user interfacing with a new sensor. This experiment focuses on measuring the system's accuracy and analyzing the underlying reasons

for any failures. These include incorrect parameters generated by the OTTER LLM or erroneous outputs produced by the OTTER Engine.

Retreival Augmentation Generation Experiment. This experiment evaluates the correctness of the Retrieval-Augmented Generation (RAG) pipeline within the OTTER LLM system. Once a sensor datasheet is fetched, it is parsed into structured markdown format and divided into split into chunks using LangChain. These chunks are then embedded and indexed using FAISS for retrieval. The goal of this experiment is to determine whether the retrieved chunks are contextually relevant to the user's prompt. We would also asses the accuracy of the internal validation prompt—*"Is this chunk helpful for the query?"*-that is a part of the OTTER LLM. We evaluate the LLM's ability to distinguish relevant chunks from irrelevant ones, using author's annotations from the original datasheet as the ground truth. This allows us to measure the effectiveness of the RAG pipeline in narrowing down meaningful content, especially given the irregular formatting of datasheet documents.

Multiple Sensor Experiment. We incrementally increase the number of connected sensors and prompt OTTER OS to configure and spawn sensor threads for each of them. This experiment validates the multi-threaded design and runtime scalability of the system, particularly the thread management and i^2c bus arbitration.

Mutliple Sensor and Hot-Swapping Plug-and-Play Experiment. The goal of this experiment is to evaluate whether OTTER OS can interface with multiple i^2c sensors simultaneously. Additionally, 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. We also monitor whether each sensor operates correctly in parallel—producing real-time output without bus collisions or timing issues. This experiment validates the OTTER OS multi-threaded design and demonstrates OTTER OS's plug-and-play capability and validates its robustness to hardware changes.

Latency Experiment. In this experiment, we evaluate the end-to-end latency of OTTER LLM from prompt to usable i^2c output across sensors of varying complexity. We break down latency into components: OEL generation, validate chunk, and retrieve chunk. We also monitor the latency of Vector DB load from local file in order to understand OTTER LLM RAG latency even further. We do not evaluate the latency within the OTTER Engine or the Mbed OS since it is negligible compared to the time required for LLM-driven code generation and datasheet processing. Moreover, this stage involves standard i^2c interfacing

encapsulated within OTTER's multithreaded engine, which behaves similarly to conventional i^{2}c operations. As such, its latency is consistent with typical i^{2}c communication and beyond the scope of control or optimization in this work.

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 i^{2}c bus via a breadboard, allowing multiple sensors to communicate over the same bus. Each i^{2}c 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 i^{2}c 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 Evaluation

```
===== 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
Running Sensor Thread: MCP9808
```

Figure 4.1: MCP9808 Sensor Struct and Output

Figure 4.1 shows the correct output of a OTTER compatible sensor, MCP9808. 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 i^{2}c 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

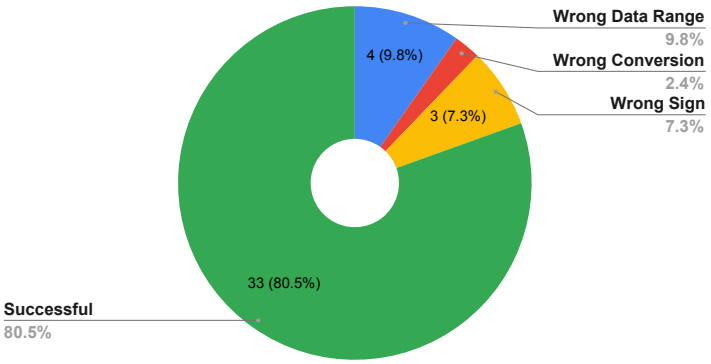

Figure 4.2: OTTER OS Accuracy

This experiment evaluates the end-to-end functional accuracy of the OTTER OS pipeline, from prompt-based sensor configuration to real-time i^2c data acquisition and output. A total of 40 evaluation runs were conducted, with 10 runs performed for each of four i^2c 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- -25.0°C when the expected result was 25.0°C. 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

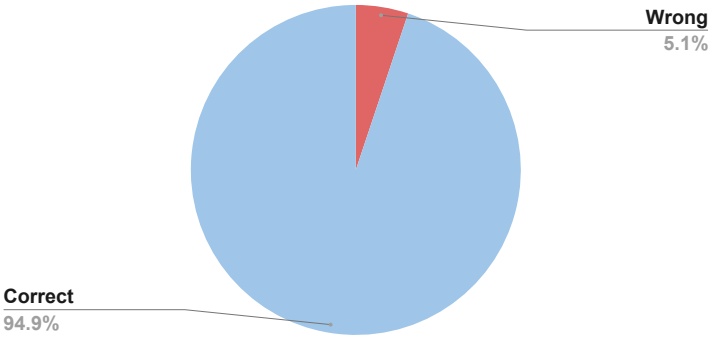

Figure 4.3: Chunk Validation Accuracy

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 Evaluation

```
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
```

Figure 4.4: i^{2}c Sensors (AHT20, SHT31, TMP102, MCP9808) Concurrent Sensor Reading Output

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 i^{2}c sensors connected simultaneously to a single i^{2}c bus. As seen in figure 4.4, 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 i^{2}c devices concurrently without conflict or data loss.

```
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

The experiment also tested OTTER's plug-and-play functionality through the use of the `NEW_SENSOR` and `SENSOR_REMOVE` commands. In figure 4.5, TMP102 sensors were added, removed, then added again dynamically during runtime, without requiring a system reboot. We then tested this on all four sensors with different combinations. The system correctly

handled thread creation, memory allocation, and i^2c initialization for new sensors, as well as safe termination and deallocation for removed sensors. This 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

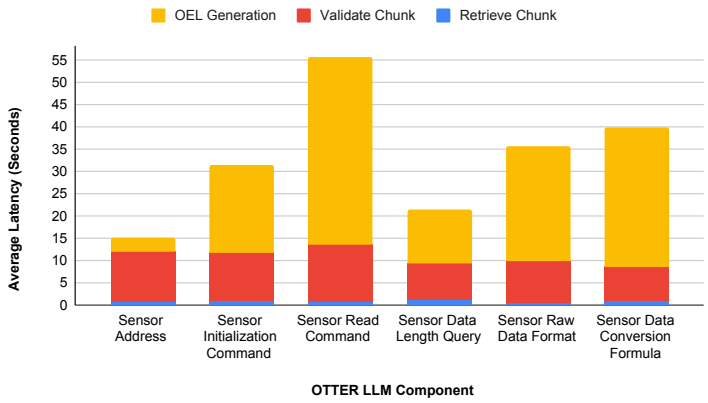

Figure 4.6: Average Latency of OTTER OS LLM Component

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.

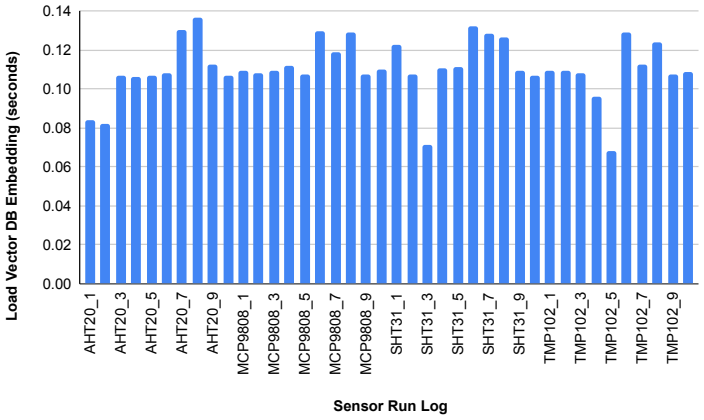

Figure 4.7: Load Vector DB Embeddings from Local File

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.

4.6 OTTER LLM Intelligence Evaluation

This evaluation is a qualitative analysis of the system meant to understand how the OTTER LLM actually provides intellectual reasoning in sensor interfacing. The OTTER LLM is the component that injects intelligence in the OTTER OS, differentiating it from other Embedded OS or sensor interfacing techniques. We can observe this intelligence by looking at the generated OEL code from 2 independent runs to interface with MCP9808 sensor. In two of the run, the same chunk context was given: *1. lower bytes, the upper byte must be right-shifted by 4 bits (or multiply by 2 [4]) and the lower byte must be leftshifted 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).* The first run generated `SCALE_FORMAT = (0: "X 0.0625 *");`. The second run generated `SCALE_FORMAT = (0: "X 16 /");`. Both of the runs produce the correct output, but the first run used multiplication with 2^{-4}, whereas the second run used division with 2^{4}. Although these approaches are mathematically equivalent,

their structural differences indicate a non-deterministic behavior in the OTTER LLM's output generation.

Additionally, the intelligence of the OTTER LLM is also observed in a more complex scenario. We can observe this by looking at the generated OEL code from 2 independent runs to interface with TMP102 sensor. In two of the run, the same chunk context was given: *1. Example: (50°C) / (0.0625°C / LSB) = 800 = 320h = 0011 0010 0000 To convert a positive digital data format to temperature: 1. Convert the 12-bit, left-justified binary temperature result, with the MSB = 0 to denote a positive sign, to a decimal number. 2. Multiply the decimal number by the resolution to obtain the positive temperature. Example: 0011 0010 0000 = 320h = 800 × (0.0625°C / LSB) = 50°C 2. To convert a negative digital data format to temperature: 1. Generate the twos compliment of the 12-bit, left-justified binary number of the temperature result (with MSB = 1, denoting negative temperature result) by complementing the binary number and adding one. This represents the binary number of the absolute value of the temperature. 2. Convert to decimal number and multiply by the resolution to get the absolute temperature, then multiply by –1 for the negative sign.* The first run generated `SCALE_FORMAT = (0: "X 2047 & X 2048 & - 0.0625 *");` The second run generated `SCALE_FORMAT = (0: "X X 11 >> 4096 * - 0.0625 *");` Both of the runs produce the correct output, but the first run used bitmask and arithmetic, whereas the second run used bitshift and arithmetic. Although these approaches are mathematically equivalent, their structural differences indicate a non-deterministic behavior in the OTTER LLM's output generation.

This result suggests that the OTTER LLM does not simply repeat a fixed answer based on what is directly stated in the datasheet or retrieved chunk. Instead, it shows the ability to reason based on the context. Even though the datasheet did not specify the exact implementation method, the model was still able to produce a correct solution by understanding the rules presented. This shows that the OTTER LLM is not just copying from memory, but is able to "think through" the problem and arrive at an answer using its internal reasoning. As a result, different correct answers may look different in structure, but still give the same outcome. This flexibility is important because it means the model can work even when the documentation does not perfectly match a fixed template.

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 i^2c 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 RAG LLM (OTTER LLM), a domain-specific language (OTTER Embedded Language), and a modular execution engine (OTTER Engine) running on a multithreaded Embedded OS.

Through detailed system design and experimentation, we demonstrated that OTTER OS can support seamless integration with multiple i^2c 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 i^2c 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 is needed to expand its expressiveness. Future iterations should support additional functionalities such as specifying sensor resolution, sampling intervals (e.g., every 10 seconds), and other operational parameters that are commonly required for real-world sensor applications.

Lastly, improving the functional accuracy of OTTER OS remains a top priority. To enhance its generalizability, it is essential to strengthen its robustness against datasheet inconsistencies, non-standard formatting, and ambiguous register descriptions—common challenges in real-world sensor documentation.

In summary, OTTER OS demonstrates a promising step toward rethinking sensor interfacing in embedded systems—shifting from a static, code-heavy process to a dynamic, prompt-driven paradigm powered by LLMs. By bridging natural language input with runtime sensor configuration, OTTER OS lowers the barrier to entry, accelerates prototyping, and opens new possibilities for intelligent, adaptive sensor networks. While challenges remain, the results presented in this thesis validate the feasibility and effectiveness of combining LLM reasoning, retrieval-augmented generation, and embedded execution into a cohesive system architecture.

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

REFERENCES

[1] Jan Schlichter. "Ph.D. Forum: Leveraging Industrial Wireless Sensor Networks for Energy-Efficient HVAC System Operations". In: *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. SenSys '24. Hangzhou, China: Association for Computing Machinery, 2024, pp. 912–913. ISBN: 9798400706974. DOI: 10.1145/3666025.3699664. URL: https://doi.org/10.1145/3666025.3699664.

[2] Yujing Zhang et al. "Poster: FlexibleBP: Blood Pressure Monitoring Using Wrist-worn Flexible Sensor". In: *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. SenSys '24. Hangzhou, China: Association for Computing Machinery, 2024, pp. 875–876. ISBN: 9798400706974. DOI: 10.1145/3666025.3699415. URL: https://doi.org/10.1145/3666025.3699415.

[3] Nagarjun Bhat et al. "ZenseTag: An RFID assisted Twin-Tag Single Antenna COTS Sensor Interface". In: *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. SenSys '24. Hangzhou, China: Association for Computing Machinery, 2024, pp. 336–350. ISBN: 9798400706974. DOI: 10.1145/3666025.3699342. URL: https://doi.org/10.1145/3666025.3699342.

[4] Orhan Konak et al. "A Real-time Human Pose Estimation Approach for Optimal Sensor Placement in Sensor-based Human Activity Recognition". In: *Proceedings of the 8th International Workshop on Sensor-Based Activity Recognition and Artificial Intelligence*. iWOAR '23. Lübeck, Germany: Association for Computing Machinery, 2023. ISBN: 9798400708169. DOI: 10.1145/3615834.3615848. URL: https://doi.org/10.1145/3615834.3615848.

[5] Lina Han, Haosu Shi, and Dongdong Xiong. "Design of Crop Environmental Monitoring System Based on ZigBee". In: *Proceedings of the 2024 9th International Conference on Cyber Security and Information Engineering*. ICCSIE '24. Association for Computing Machinery, 2024, pp. 454–460. ISBN: 9798400718137. DOI: 10.1145/3689236.3689271. URL: https://doi.org/10.1145/3689236.3689271.

[6] Yilang Qin et al. "Design and application of smart agriculture IoT platform based on microservice architecture". In: *Proceedings of the 2024 4th International Conference on Internet of Things and Machine Learning*. IoTML '24. Association for Computing Machinery, 2024, pp. 253–259. ISBN: 9798400710353. DOI: 10.1145/3697467.3697654. URL: https://doi.org/10.1145/3697467.3697654.

[7] Aim Lay-Ekuakille and Subhas Chandra Mukhopadhyay. *Wearable and Autonomous Biomedical Devices and Systems for Smart Environment: Issues and Characterization*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 364215686X.

[8] Tianyu Zhang et al. “Time-Sensitive Networking (TSN) for Industrial Automation: Current Advances and Future Directions”. In: *ACM Comput. Surv.* 57.2 (Oct. 2024). ISSN: 0360-0300. DOI: 10.1145/3695248. URL: https://doi.org/10.1145/3695248.

[9] Mouser Electronics. *I^2C Sensors*. Accessed: 2025-04-08. URL: https://www.mouser.com/c/sensors/?interface%20type=I2C.

[10] Zhixin Xie et al. “BitDance: Manipulating UART Serial Communication with IEMI”. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID ’23. Hong Kong, China: Association for Computing Machinery, 2023, pp. 63–76. ISBN: 9798400707650. DOI: 10.1145/3607199.3607249. URL: https://doi.org/10.1145/3607199.3607249.

[11] Hibiki Shinozaki and Akira Yamawaki. “Hardware Implementation of Calibration Data Loading in Device Driver for an SPI Peripheral”. In: *Proceedings of the 2024 6th International Electronics Communication Conference*. IECC ’24. Fukuoka, Japan: Association for Computing Machinery, 2024, pp. 19–23. ISBN: 9798400717598. DOI: 10.1145/3686625.3686629. URL: https://doi.org/10.1145/3686625.3686629.

[12] Charalampos Patsianotakis et al. “I2C Bridge on FPGA to Facilitate Vast Connection of Heterogeneous IoT Devices in Agriculture Applications”. In: *Proceedings of the 2024 8th International Conference on Algorithms, Computing and Systems*. ICACS ’24. Association for Computing Machinery, 2025, pp. 142–149. ISBN: 9798400718304. DOI: 10.1145/3708597.3708619. URL: https://doi.org/10.1145/3708597.3708619.

[13] Adafruit Industries. *Adafruit Industries - Unique & fun DIY electronics and kits*. Accessed: 2025-04-08. URL: https://www.adafruit.com/.

[14] Pololu Robotics and Electronics. *Pololu Robotics and Electronics*. Accessed: 2025-04-08. URL: https://www.pololu.com/.

[15] Kasimir Aula et al. “Evaluation of Low-cost Air Quality Sensor Calibration Models”. In: *ACM Trans. Sen. Netw.* 18.4 (Dec. 2022). ISSN: 1550-4859. DOI: 10.1145/3512889. URL: https://doi.org/10.1145/3512889.

[16] Silicon Laboratories. *Sensors*. Accessed: 2025-04-08. URL: https://www.silabs.com/sensors.

[17] Adafruit Industries. *Adafruit AHTX0 Library*. Accessed: 2025-04-08. 2019. URL: https://github.com/adafruit/Adafruit_AHTX0.

[18] Adafruit Industries. *Adafruit MCP9808 Library*. Accessed: 2025-04-08. 2014. URL: https://github.com/adafruit/Adafruit_MCP9808_Library.

[19] Adafruit Industries. *Adafruit SHT31 Library*. Accessed: 2025-04-08. 2016. URL: http://github.com/adafruit/Adafruit_SHT31.

[20] SparkFun Electronics. *SparkFun TMP102 Arduino Library*. Accessed: 2025-04-08. 2016. URL: https://github.com/sparkfun/SparkFun_TMP102_Arduino_Library.

[21] OpenAI. *GPT-4o: OpenAI's New Multimodal Model*. Accessed: 2025-04-08. May 2024. URL: https://openai.com/index/gpt-4o.

[22] Meta AI. *Introducing Llama 3: The Next Generation of Open Foundation Models*. Accessed: 2025-04-08. Apr. 2024. URL: https://ai.meta.com/llama.

[23] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997.

[24] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401.

[25] Rick Merritt. *What Is Retrieval-Augmented Generation, aka RAG?* Accessed: 2025-04-08. Jan. 2025. URL: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/.

[26] Elmin Marevac et al. "Framework Design for the Dynamic Reconfiguration of IoT-Enabled Embedded Systems and "On-the-Fly" Code Execution". In: *Future Internet* 17.1 (2025). ISSN: 1999-5903. DOI: 10.3390/fi17010023. URL: https://www.mdpi.com/1999-5903/17/1/23.

[27] Florian Schade et al. "Dynamic Partial Reconfiguration for Adaptive Sensor Integration in Highly Flexible Manufacturing Systems". In: *Procedia CIRP* 107 (2022). Leading manufacturing systems transformation – Proceedings of the 55th CIRP Conference on Manufacturing Systems 2022, pp. 1311–1316. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2022.05.150. URL: https://www.sciencedirect.com/science/article/pii/S2212827122004346.

[28] William Richards, John Selker, and Chet Udell. "Loom: A Modular Open-Source Approach to Rapidly Produce Sensor, Actuator, Datalogger Systems". In: *Sensors* 24.11 (2024). ISSN: 1424-8220. DOI: 10.3390/s24113466. URL: https://www.mdpi.com/1424-8220/24/11/3466.

[29] Zachary Englhardt et al. "Exploring and Characterizing Large Language Models for Embedded System Development and Debugging". In: *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. CHI EA '24. Honolulu, HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703317. DOI: 10.1145/3613905.3650764. URL: https://doi.org/10.1145/3613905.3650764.

[30] Huanqi Yang et al. *EmbedGenius: Towards Automated Software Development for Generic Embedded IoT Systems*. 2024. arXiv: 2412.09058 [cs.SE]. URL: https://arxiv.org/abs/2412.09058.

[31] Embedd.it. *Embedd.it - Software Integration and Testing of Chips*. Accessed: 2025-04-08. URL: https://embedd.it/.

[32] Anysphere Inc. *Cursor: The AI Code Editor*. Accessed: 2025-04-09. URL: https://www.cursor.com/.

[33] Philip Levis et al. "TinyOS: An operating system for wireless sensor networks". In: *Ambient Intelligence*. Springer, 2005, pp. 115–148.

[34] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors". In: *29th Annual IEEE International Conference on Local Computer Networks*. IEEE. 2004, pp. 455–462.

[35] Chih-Chieh Han et al. "A dynamic operating system for sensor nodes". In: *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*. MobiSys '05. Seattle, Washington: Association for Computing Machinery, 2005, pp. 163–176. ISBN: 1931971315. DOI: 10.1145/1067170.1067188. URL: https://doi.org/10.1145/1067170.1067188.

[36] Zephyr Project. *Zephyr Project Official Website*. Accessed: 2025-04-08. URL: https://www.zephyrproject.org/.

[37] Nicholas H Tollervey. *Programming with MicroPython: embedded programming with microcontrollers and Python*. " O'Reilly Media, Inc.", 2017.

[38] Damien George. *MicroPython - Python for Microcontrollers*. https://micropython.org. Accessed: 2025-04-07. 2013.

[39] Ulrich Ter Horst, Hagen Hasberg, and Stephan Schulz. "MicroPython-based sensor node with asymmetric encryption for ubiquitous sensor networks". In: *2021 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*. IEEE. 2021, pp. 1–6.

[40] Sungjoo Yoo and Ahmed Jerraya. "Introduction to Hardware Abstraction Layers for SoC". In: Feb. 2003, pp. 336–337. ISBN: 0-7695-1870-2. DOI: 10.1109/DATE.2003.1253629.

[41] James Smith. *I2C Abstraction Layer*. Accessed: 2025-04-09. 2015. URL: https://github.com/loopj/i2c-hal.

[42] Zhaoyun Zhang and Jingpeng Li. "A review of artificial intelligence in embedded systems". In: *Micromachines* 14.5 (2023), p. 897.

[43] Sinan Sabree and Alameer Albadrani. *OpenAI as a Tool for Programming Embedded Systems*. 2024.

[44] GitHub and OpenAI. *GitHub Copilot: Your AI Pair Programmer*. Accessed: 2025-04-09. 2021. URL: https://github.com/features/copilot.

[45] Replit. *Replit Ghostwriter: Your Partner in Code*. Accessed: 2025-04-09. 2022. URL: https://blog.replit.com/ghostwriter.

[46] Zachary Englhardt et al. *Exploring and Characterizing Large Language Models For Embedded System Development and Debugging*. 2023. arXiv: 2307.03817 [cs.SE]. URL: https://arxiv.org/abs/2307.03817.

LIST OF FIGURES

LIST OF TABLES