LLMind 2.0: Distributed IoT Automation with Natural Language M2M Communication and Lightweight LLM Agents

Yuyang Du, *Student Member, IEEE*, Soung Chang Liew, *Fellow, IEEE*,

Abstract*—Recent advances in large language models (LLMs) have sparked interest in their application to IoT and automation systems, particularly for facilitating device management through natural language instructions. However, existing centralized approaches face significant scalability challenges when managing and coordinating the collaboration between IoT devices of diverse capabilities in large-scale heterogeneous IoT systems. This paper introduces LLMind 2.0, a distributed IoT automation framework that addresses the scalability challenges through lightweight LLM-empowered device agents via natural language-based machine-to-machine (M2M) communication. Unlike previous LLM-controlled automation systems that rely on a centralized coordinator to generate device-specific code to be executed on individual devices, LLMind 2.0 distributes intelligence across individual devices through lightweight LLMs embedded in IoT devices. The central coordinator translates human instructions into simple subtasks described in natural human language, which are then processed by device-specific agents to generate device-specific code locally at the associated devices. This approach transcends device heterogeneity barriers by using natural language as a unified communication medium, enabling seamless collaboration between devices from different manufacturers. The system incorporates several key innovations: a Retrieval-Augmented Generation (RAG) mechanism for accurate subtask-to-API mapping, fine-tuned lightweight LLMs for reliable code generation, and a finite state machine-based task execution framework. Experimental validation in multi-robot warehouse scenarios and real-world WiFi network deployments demonstrates significant improvements in scalability, reliability, and privacy protection compared to the centralized approach. The distributed architecture enables parallel processing, reduces coordinator's computational burden, and facilitates natural collaborative behaviors among IoT devices.

***Index Terms*—xxxx, xxxxxxxxx, xxxxxxxxxxxx, xxxx, xxxxxxxxxxx**

I. Introduction

Recent breakthroughs in generative AIs have attracted significant interest in applying [1], [2] large language models (LLMs) to facilitate efficient system automation and device management [3]–[5] in complex IoT systems consisting of collaborative IoT devices of diverse capabilities. In this line of research, LLM serves as the central system coordinator, translating human commands into machine executable codes and assigning the codes to IoT devices to execute complex tasks involving the collaborative efforts of multiple IoT devices. Although this technical solution has proven effective for small-scale IoT systems, it has inherent scalability limitations for larger-scale systems with device heterogeneity.

This paper explores the latest technical trend in lightweight embedded LLMs and attempts to address the scalability problem by introducing distributed device agents into IoT management and automation systems empowered by LLMs. The integration of device agents opens new possibilities for the communication methods between the system coordinator and distributed devices: specifically, a new type of machine-to-machine (M2M) communication that leverages natural language as the unified interaction medium between the coordinator and IoT devices. As a replacement for the rigid, code-based interaction pattern – where the coordinator generates device-specific code for the diverse devices to execute – the natural language-based M2M interface transcends the barriers of device heterogeneity, allowing devices from different manufacturers (or even supported by different programming languages) to communicate and collaborate using a unified human language. Code generation is entrusted to device agents that know the languages of the associated devices intimately. Furthermore, as the natural language-based M2M interface shifts language-code transformation tasks from the central coordinator to distributed device agents, the coordinator's workload could be significantly reduced so that it can easily manage a larger-scale system. Additionally, the reliability of code generation is also enhanced with LLM-supported device agents that are specifically fine-tuned for the code generation task. Meanwhile, compared with previous solutions where a device's information is uploaded to a cloud-based LLM coordinator, the device's privacy and safety risk can be eliminated with the new natural language-based M2M interface, given that all device agents are locally deployed.

In short, this paper explores M2M communication using human natural language rather than machine languages. An added advantage is transparency, in which humans can easily understand machine interactions in automated systems when needed through logs written in natural language.

The rest of the introduction elaborates on the above arguments with an overview. Subsection A articulates the limita-

Manuscript received 6 March 2024; revised 22 September 2024, and 2 January 2025; accepted 20 Feberuary 2025. Date of publication xx xxx 2025; date of current version xx xxx 2025. The review of this article was coordinated by Dr. Mugen Peng.

This work was supported in part by the General Research Funds (Project No. 14200221) established under the University Grant Committee of the Hong Kong SAR, China, and in part by CUHK Direct Grant (Project Code: 405524).

The research work described in this paper was conducted in the JC STEM Lab of Advanced Wireless Networks for Mission-Critical Automation and Intelligence funded by The Hong Kong Jockey Club Charities Trust.

Y. Du and S. C. Liew are with the Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong SAR, China (e-mail: {dy020, soung}@ie.cuhk.edu.hk). S. C. Liew is the corresponding author.

tions of conventional systems, and subsection B introduces our solution. Novel challenges arising from the implementation of the proposed system, as well as how they are addressed in our system, are highlighted in subsection C. This is followed by a summary of our contributions in subsection D.

A. Limitations of existing systems

Previously reported approaches relied heavily on an all-knowing LLM coordinator to translate natural language instructions, such as those provided by human users, into device-executable code. In [3], authored by us, proposed a task-oriented human-to-machine communication framework, where an LLM acts as a centralized coordinator to orchestrate IoT devices in executing complex tasks. This framework, known as LLMind, uses a language-code transformation approach, with the LLM translating verbal instructions into a finite-state machine (FSM) representation as an intermedia, which is then transformed into executable codes for diverse devices, possibly from different manufacturers. The components in the frameworks of other studies could also potentially serve as the centralized code generator. For example, [6] developed an instruction-tuning dataset to train an open-source LLM to execute human instructions through application APIs, while [7] created a dataset of API documentation to fine-tune the LLM, improving its understanding of API structures and syntax. Additionally, [8] fine-tuned a LLaMA model to generate accurate API function calls and adapted to test-time document changes by integrating a document retriever.

While effective for small-scale systems, the reliance on a centralized coordinator and code-based M2M interface introduces significant scalability challenges as the diversity and number of devices increase:

1) **API Adaptation Complexity:** Proprietary APIs from diverse devices are often written in different programming languages and incorporate device-specific specifications. This heterogeneity makes it increasingly complex for the LLM coordinator to generate accurate executable code as device diversity grows. Additionally, integrating new devices requires ensuring compatibility with existing ones, further exacerbating system complexity.
2) **Coordinator Computation Bottleneck:** As the number of devices grows, the centralized coordinator must generate device-specific executable code for many devices simultaneously. This creates significant computational strain, leading to inefficiencies and potential delays.

B. Our Solution: LLMind 2.0

We propose to leverage lightweight LLM-empowered IoT device agents, which use natural language as M2M communication interfaces, to address the scalability problem. In this approach, a conventional large-scale LLM acts as the central coordinator that translates natural language instructions - such as those provided by a human - into a series of subtasks, also specified in natural language. The coordinator then transmits these natural language subtask specifications, rather than executable code, to device-specific agents. These agents, realized with embedded lightweight LLMs in IoT devices (also known as embedded LLMs), are responsible for interpreting the natural language-based subtask instructions and generating executable code tailored to their respective devices' proprietary APIs.

This distributed approach fundamentally changes the computation and communication paradigm in existing systems: rather than relying on the LLM coordinator to handle device-specific code generation for all devices, the devices themselves, through their device agents, become active participants in the translation process. Using natural language as the medium of communication between the central coordinator and device agents gives rise to several key advantages:

1) **Enhanced Scalability:** The computational burden is offloaded from the centralized LLM coordinator, as device-specific agents handle the translation from natural language to executable code. This reduces the computation strain on the central coordinator as the number of devices grows.
2) **Improved Flexibility:** Device-specific agents are designed to interpret and process subtasks specific to their associated devices only, making it easier to integrate new devices with proprietary APIs into the system without requiring extensive updates to the LLM coordinator.
3) **Facilitation of Human-Readable M2M Interactions:** By using natural language as the communication medium, this approach paves the way for devices to interpret and even share instructions in human-readable form, enabling more transparent M2M communication.
4) **Cost-Efficient Device Agents:** The device agents do not need to rely on costly, general-purpose large language models. Instead, smaller, task-specific language models can be employed as agents since they only need expertise specific to their respective devices, rather than maintaining general knowledge. This reduces computational requirements and operational costs for individual devices.
5) **Privacy:** In previous systems [3], [6], [8], code generations require device-specific information, such as API interface or detailed descriptions of the environment that the device works in. By shifting the duty from cloud-based large-scale LLMs (such as GPT-4 used in LLMind [3]) to locally deployed lightweight device agents, the system's safety and privacy concerns can be well eliminated.
6) **Parallel Code generation:** the distributed architecture enables the parallel translation of multiple natural language subtasks by different device agents into device-executable code, significantly enhancing system efficiency.

Through this distributed agent-based approach, we aim to overcome the scalability limitations of centralized IOT automation systems while enabling efficient, flexible, and privacy-protecting IoT device management with the natural language-based M2M interactions between devices and the central coordinator. Building upon the original LLMind framework in [3], which is henceforth referred to as LLMind 1.0 for distinction, this paper puts forth LLMind 2.0, a fundamentally redesigned successor with the integration of all the above

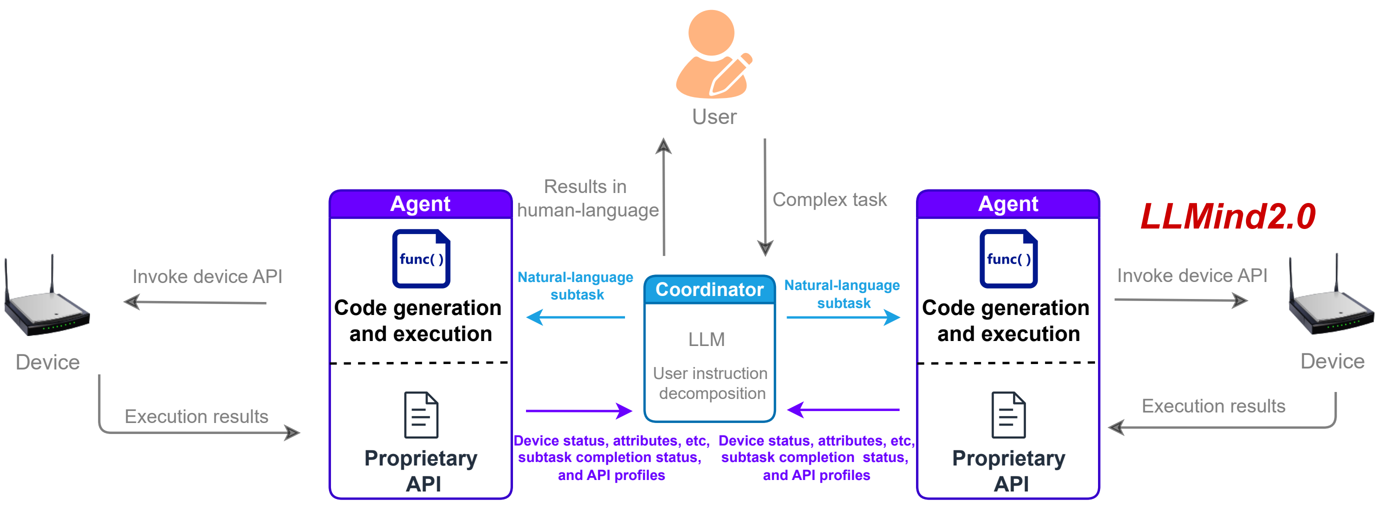

User
Results in human-language
Complex task
LLMind2.0
Agent
func( )
Code generation and execution
Proprietary API
Coordinator
LLM
User instruction decomposition
Natural-language subtask
Device status, attributes, etc, subtask completion status, and API profiles
Invoke device API
Execution results
Device

Figure 1: The depicted architecture is designed to address the scalability challenges associated with managing and interacting with a vast array of devices from different vendors, each with its own proprietary interfaces. These device-specific agents, equipped with embedded LLMs significantly smaller than the coordinator's LLM, are proficient in natural language processing and possess intimate knowledge of their respective devices' proprietary interfaces. By focusing solely on their specific devices' subtasks for code generation, these agents also reduce the design complexity of the overall system.

features. Fig. 1 illustrates the LLMind 2.0 system. At its core is a central coordinator containing an LLM that interprets human instructions and translates them into natural language subtasks. These subtasks are then executed by various devices through their respective agents.

C. Technical Innovations and Experimental Validations

Although the new framework of LLMind 2.0 has many compelling benefits, the design of the device agents in Fig. 1 presents non-trivial challenges. The agents must correctly interpret the natural language subtasks provided by the coordinator to avoid inaccurate subtask executions. Misinterpretation could stem from two primary reasons. The first is the mistaken selection of the device API function call that mismatches the natural language described subtask. Further, even when the correct API function is selected, the code generation process itself may be error-prone. To address potential translation errors, the following techniques have been adapted to our agent system.

1) **Retrieval-Augmented Generation (RAG) for Accurate Mapping of Subtasks to Device APIs:** To ensure accurate mapping between subtasks and proprietary device API functions, we implement a RAG scheme, a proven technique that leverages external knowledge to enhance LLM performance. For each device agent, we construct an external knowledge base containing a device's proprietary APIs, convert the given natural language-based subtask description into vector embedding, and retrieve the most relevant device API function according to the semantic similarity between 1) the vector embedding of the given subtask description, and 2) the vector embedding of the device's APIs description. The RAG scheme will be introduced in Section III-B in detail.
2) **Finite State Machine (FSM) Based Code Generation with LLM Fine-Tuning:** Previous research [3] has introduced an FSM-based code generation framework with significantly enhanced reliability. This paper inherits the scheme and further improves it by fine-tuning the device-agent LLMs. The FSM-based code generation pipeline helps the agent to avoid generating incomplete or syntactically incorrect code, while the fine-tuning process enhances the agent's ability to accurately extract input argument values from the natural language description of the subtask, ensuring the precise execution API calls. Section III-A introduces the FSM-based code generation pipeline, which includes the following five states: 1) start, 2) preprocessing in progress, 3) function-call result pending, 4) postprocessing in progress, and 5) end. Section III-C explains the fine-tuning process.

The above mechanisms ensure accurate execution of natural language subtasks by devices and address the complexity of M2M communication across a diverse array of devices. To validate the proposed approaches, we implemented LLMind 2.0 in Python on x86-based PCs and conducted the following two experiments:

Experiment #1: In this experiment, a warehouse manager uses natural language to instruct mobile robots to collaboratively search for vacancies across multiple shelves. Through this experiment, we compare the performance of this paper's distributed scheme with the centralized scheme in [3] on executing the task of with multiple heterogeneous robots involved.

Experiment #2: In this experiment, multiple IoT devices deployed over a Wi-Fi network perform two challenging networking tasks: 1) Quality of Service (QoS) adjustment using the Enhanced Distributed Coordination Function (EDCF) [9], and 2) interference detection and mitigation. Through this experiment, we demonstrate LLMind 2.0's distributed intelligence. The experiment demonstrates the devices successfully completing both networking tasks through competition and cooperation.

D. Summary of Contributions

With the development of LLMind 2.0, a system with distributed LLM-empowered agents that enables seamless natural-language M2M communication across a large number of devices, this paper makes the following contributions:

1) **Distributed Device Agents Powered by Advanced Lightweight LLMs:** We have developed embedded agents for IoT devices with advanced lightweight LLMs. The introduction of these distributed agents significantly enhances the scalability, efficiency, flexibility, and privacy protection of state-of-the-art LLM-controlled IoT automation systems.
2) **Pioneering M2M Communication with Natural Language Interface:** We pioneers the idea of natural-language M2M communication, facilitating the seamless integration of heterogeneous device agents. To the best of our knowledge, LLMind 2.0 is the first ecosystem that investigates natural-language M2M communication interfaces across diverse devices and demonstrates the feasibility with distributed LLM-powered agents.
3) **A Reliable Code Generation Pipeline with RAG-based API Function matching and Model Fine-Tuning:** To achieve error-free translation from a subtask's language description to associated device executable code, the distributed device agent in LLMind 2.0 integrates a RAG scheme to facilitate accurate mapping of natural-language subtasks to device API functions. Further, the agent LLM inherits and expands the FSM-based code generation pipeline from LLMind 1.0, which was originally developed for the cloud-based LLM working as the system's central coordinator. We have also enhanced the reliability of code generation by fine-tuning the lightweight agent LLM with the ability to precisely extract the input-argument values from the subtask description, which is a vulnerable point we previously noticed in the FSM-based code generation pipeline.

To support further research in the community, we have open-sourced the training dataset and code used to fine-tune the agent LLMs at: https://github.com/1155157110/LLMind2.0.

II. Motivation and System Overview

Subsection A presents a case study in a multi-robot collaboration scenario to illustrate the scalability challenges inherent in the conventional LLM-managed IoT automation system, which relies on a centralized LLM to generate machine-executable codes as the medium of "coordinator-device" interactions. Subsection B summarizes the lessons learned from the case study and motivates the design principle of LLMind 2.0, addressing the scalability problem as a central theme.

A. Motivations illustrated with a case study

To better understand the limitations of the current LLMind 1.0 system with code-based M2M communication interface, let us consider a warehouse scenario with N shelves and N mobile robots of different brands, with each robot equipped with a camera to identify vacant positions. The warehouse manager issued the following instruction to the multi-robot system: ”*Please help check if there are vacant spaces on the shelves.*”

In executing this task, a conventional uncoordinated IoT system requires manual task assignments (i.e., individually programming robots for the navigation and vacancy searching task). LLMind 1.0 significantly reduces labor costs while enhancing operational efficiency with the assistance of LLMs. Going further, we are interested in the answers to the following questions: 1) *Is LLMind 1.0 the best solution?* 2) *What constitutes a good technical solution for automatic IoT control systems like LLMind 1.0?*

Our arguments are as follows: in modern warehouse settings, where the number of shelves can be substantial, a good solution must enable the multi-robot system to improve search efficiency through parallel collaboration while maintaining a high task success rate. Specifically:

1) **Scalability:** The total time required for the shelf vacancy search task should not increase significantly with the number of shelves N, ideally growing only sub-linearly.
2) **Reliability:** The success rate of the shelf vacancy search task should remain consistently high, even as N increases.

Limitations of Current Practice: With the above in mind, we adopt LLMind 1.0 as a baseline. In the multi-robot shelf vacancy search experiment, LLMind 1.0 uses code as the M2M interface between the central coordinator and IoT devices. For better technical focus, here we do not present implementation details of the baseline. We refer readers to Appendix A of this paper for a comprehensive introduction to the system design in [3].

We evaluate LLMind 1.0's scalability by conducting tests with N ranging from 1 to 10. As illustrated in Fig. 2, the centralized coordinator uses a serial processing pipeline to orchestrate N robots in the search for shelf vacancies: 1) planning a task for Robot 1 as an FSM; 2) generating code for Robot 1 based on the planned FSM; 3) repeating steps 1) and 2) for each subsequent robot. In step 1), to facilitate the coordinator's task planning process, the prompt provides the LLM with environment information, device APIs, and a task description [3]. The LLM coordinator then composes an FSM for each robot, with each FSM action corresponding to a device's API function (note: a detailed explanation of the prompt used in [3], as well as the FSM-based code generation process within LLMind 1.0 [3], is available in Appendix A of this paper). The coordinator does not need to wait for the execution results from one robot (i.e., performing the shelf search task) before generating code for the next robot.

Fig. 3 summarizes the elapsed time incurred by the centralized coordinator to execute the shelf vacancy search task and the success rate of code generation for various N. For each N, we conducted 100 experiments, the average results of which are shown in Fig. 3. The legend ”Entire process” refers to the total elapsed time spent on task planning and FSM-based code generation performed by the centralized coordinator.

The experimental results reveal a pronounced linear increase in time consumption for both the overall process and the code generation phase as N increases from 1 to 10. This escalating time cost becomes increasingly unsustainable as N grows,

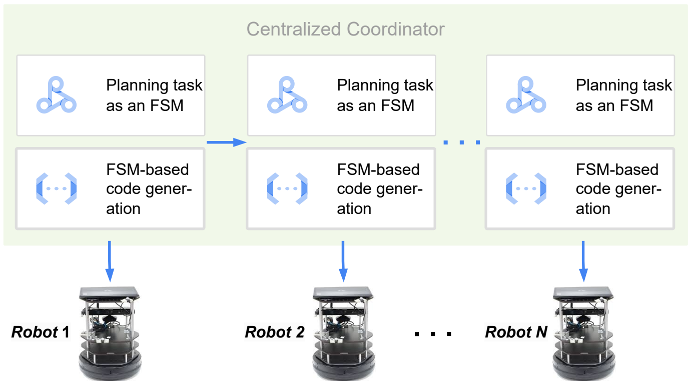


Figure 2: The processing pipeline of the centralized coordinator [3] in the shelf vacancy search task.

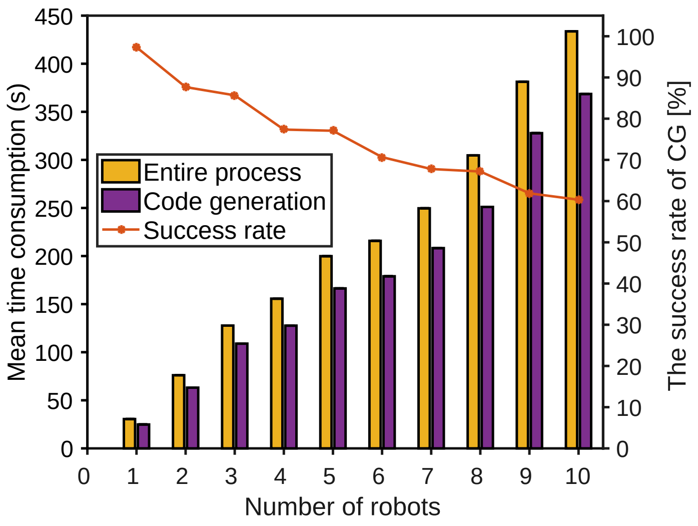


Figure 3: Overall performance of the baseline solution in the multi-robot collaboration scenario

underscoring a critical scalability issue in centralized LLM-based M2M communication schemes with code interface. Notably, the code generation phase accounts for a significant portion of the total task execution time. Further, as N increases, the success rate of code generation decreases significantly, indicating that the centralized scheme struggles to reliably generate correct executable code for larger robot fleets.

We identify three critical scalability challenges in the code-interface M2M communication pattern reported in LLMind 1.0:

1) **Challenge #1 - Inefficiency of Serialized Processing:** The task of searching for shelf vacancies is inherently parallelizable when multiple robots are available. However, the centralized LLM coordinator generates executable code sequentially for each robot, and the unparalleled code generation process becomes a bottleneck for the overall task.
2) **Challenge #2 - Coordinator Computation Bottleneck:** The centralized coordinator must generate device-specific executable codes for robots of different brands (note: in a diverse IoT system, the robots could be of different brands and may have different APIs). As N increases, the growing computational workload overwhelms the coordinator, creating a critical bottleneck.
3) **Challenge #3 - Complexity of Code Generation for Diverse Device APIs:** The heterogeneity of devices introduces significant complexity to the code generation process. As N grows, it becomes increasingly time-consuming and error-prone for the LLM coordinator to produce accurate executable code. Ensuring compatibility and reliability across diverse device APIs further exacerbates this challenge.

B. System Design Principles

The observations in subsection A lead to the following key focuses for a redesign effort:

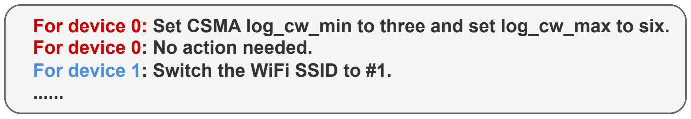
1) **Architectural Improvements:** To address Challenge #1, we can design a distributed system that decentralizes intelligence across individual devices. Distributing computational tasks ensures the overall computational time does not increase dramatically as N increases, thanks to parallel computations by distributed devices.
2) **Coordinator Optimization:** To address Challenge #2, the LLM coordinator can leverage its advanced natural language generation capabilities to directly instruct devices in natural language, rather than generating device-executable code. This shift reduces the computational burden on the coordinator, streamlining operations and improving overall efficiency.
3) **API Adaptation Enhancements:** To address Challenge #2 and Challenge #3, we propose to develop device-specific agents that can interpret natural language instructions and generate code tailored to their associated devices. This localized approach simplifies the adaptation process for proprietary APIs across diverse devices, enhancing both reliability and scalability.

With the above design principles, we now give an overview of our system design. Realization details of the LLMind 2.0 framework will be presented in Section III. In general, LLMind 2.0 addresses the scalability challenges with distributed lightweight LLM-empowered device agents that use natural language-based M2M communication. With the illustration of Fig. 1 (see Section I), we now describe the operation of our distributed system.

Coordinator: The central coordinator executes the following three consecutive steps periodically:

1) Poll and collect integrated data from each device agent, including: 1) each device's current status; 2) each device's attributes; 3) the device's API profiles, and 4) the completion status of the subtask running on the device.
2) Based on the latest device data collected, the coordinator composes a complex task consisting of multiple subtasks specified in natural language for execution by devices (Fig. 4 is an example of the natural language specification, which is used in our experiments in Section IV-B).
3) The coordinator distributes subtask specifications to device agents, who will then generate device-specific code for execution by the respective devices.

The interaction protocol between the coordinator and the device, which is described in Step 1 and Step 2, adheres to the system protocol presented in Section III-A.



For device 0: Set CSMA log_cw_min to three and set log_cw_max to six.
For device 0: No action needed.
For device 1: Switch the WiFi SSID to #1.
......

Figure 4: Examples of subtasks specified in natural language. These subtasks are concise, single-sentence instructions directed at specific devices. Each subtask corresponds to a device API function call with input arguments.

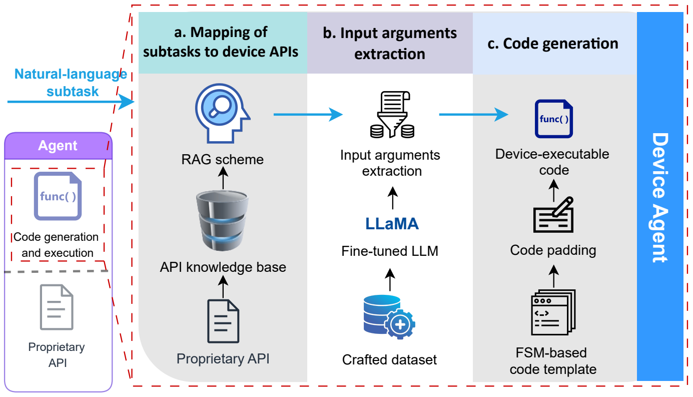
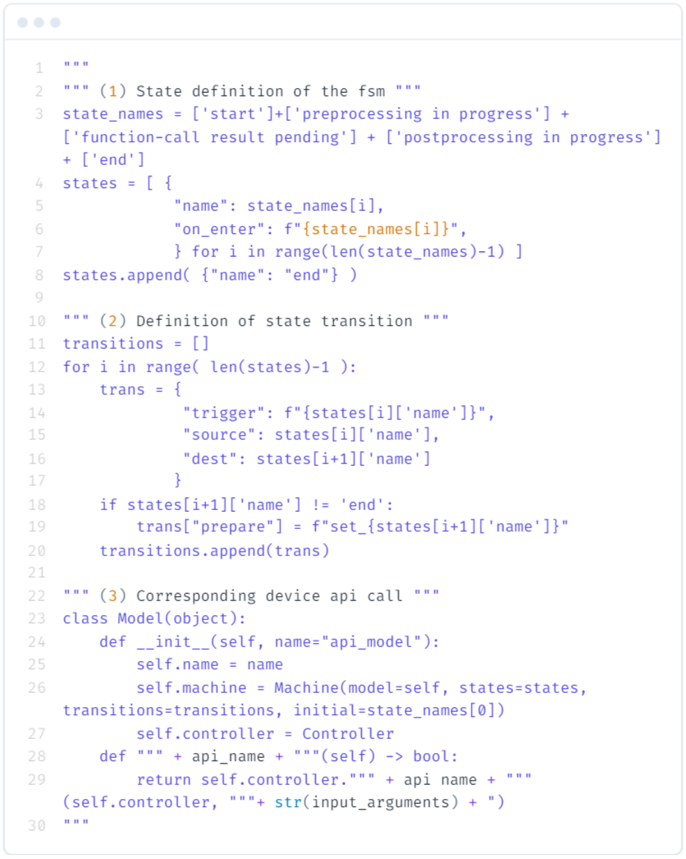


Figure 5: An example of the proposed device agent.

Device Agents: Fig. 5 shows the schematic for the device agent. As can be seen from the figure, the agent's code generation process inherits the FSM-based scheme from LLMind 1.0. The key challenge is that the scheme was previously developed for large-scale cloud-based LLMs such as GPT-4, while in LLMind 2.0, the scheme is realized with less powerful, lightweight LLMs deployed in IoT devices. This may compromise the reliability and response speed of the code generation process without a thoughtful design.

The LLMind 2.0 device agent addresses this challenge with a template-based method. Specifically, we implement the FSM-based code structure in LLMind 1.0 as a reusable code template, where there are placeholders for task-customized information such as API function names and input arguments. To illustrate, Fig. 6 shows the code template for the FSM-based code structure. This FSM is implemented with five states (see Line 3 of Fig. 6), and the operation of each state is triggered on the entry of a state (see the state definition from Line 4 to Line 8). Given a subtask instruction described in natural language, the agent LLM no longer writes the complete code, instead it only generates several lines of executable coded tailored to this specific task as a replacement of the placeholder in the template (e.g., see Line 28 and 29 in Fig. 6). This template-based method reduces the agent LLM's pressure in code generation, enabling the agent to response in a more reliable and swift manner. Meanwhile, the scheme does not harm the agent's flexibility and generalization when facing diverse tasks, given that the customized code for API function call is written according to the specific task given. See Section III-A for execution details of the five-state FSM.



```
1  """
2  """ (1) State definition of the fsm """
3  state_names = ['start']+['preprocessing in progress'] +
   ['function-call result pending'] + ['postprocessing in progress']
   + ['end']
4  states = [ {
                "name": state_names[i],
                "on_enter": f"{state_names[i]}",
                } for i in range(len(state_names)-1) ]
8  states.append( {"name": "end"} )
9
10 """ (2) Definition of state transition """
11 transitions = []
12 for i in range( len(states)-1 ):
13     trans = {
14                "trigger": f"{states[i]['name']}",
15                "source": states[i]['name'],
16                "dest": states[i+1]['name']
17               }
18     if states[i+1]['name'] != 'end':
19         trans["prepare"] = f"set_{states[i+1]['name']}"
20     transitions.append(trans)
21
22 """ (3) Corresponding device api call """
23 class Model(object):
24     def __init__(self, name="api_model"):
25         self.name = name
26         self.machine = Machine(model=self, states=states,
   transitions=transitions, initial=state_names[0])
27         self.controller = Controller
28     def """ + api_name + """(self) -> bool:
29         return self.controller.""" + api name + """
   (self.controller, """+ str(input_arguments) + ")
30 """
```

Figure 6: Code template corresponding to the five-state FSM.

With the template-based method, the only remaining problem is the precise generation of the task-specific function calls. Recall from the above discussion that there are two task-customized information that the LLM needs to consider: 1) which API function(s) should be used to complete the task, and 2) what are the input arguments of each API function. The following techniques have been developed to ensure the agent LLM can obtain the above information in a precise and prompt manner.

1) **Mapping subtasks to device-specific APIs.** To achieve this, we build a comprehensive knowledge base of customized API for each device, which helps the agent better understand the abilities of the device it is now working on. Further, for the reliable matching between the subtask's natural language description and the corresponding API that is required for completing the subtask, we implement an RAG-based mapping mechanism to achieve accurate matching. See Section III-B for details.
2) **Extract input-argument values required for an API function.** To do so, the device agent needs to analyze the natural-language subtask specification again to extract the input arguments for the function. To ensure reliable and swift argument extraction, i.e., the agent can produce the input argument of an API call based on the function name and the natural-language subtask specification in a precise and timely manner, we fine-tune the agent LLM with a dataset of instruction-argument pairs we have created.

We highlight that the extraction task is non-trivial for the agent LLM, considering potential unreliable performance of existing lightweight LLMs without fine-tuning. See Section III-C for details.

III. Design and Implementation

Subsection A explains the interaction between the coordinator and the device agent, as well as how the device agent executes an assigned subtask. Subsections B and C concern the design of device agents, presenting implementation details about the RAG-based subtask-API mapping mechanism and the agent fine-tuning process, respectively.

A. System Protocol Designs

This subsection describes the interaction protocol between the coordinator and a device agent. Specifically, we investigate a fault-tolerant periodic protocol with time-outs that is deadlock-free.

Coordinator Protocol: Fig. 7 shows the coordinator's state transition diagram with two threads of execution. The ellipses are the states, E is the event that triggers the exit from a state, and A is the action performed by the thread in between the transition from one state to another state. When a human user issues an instruction, the first thread records the input for later action(s) to be performed by the second thread.

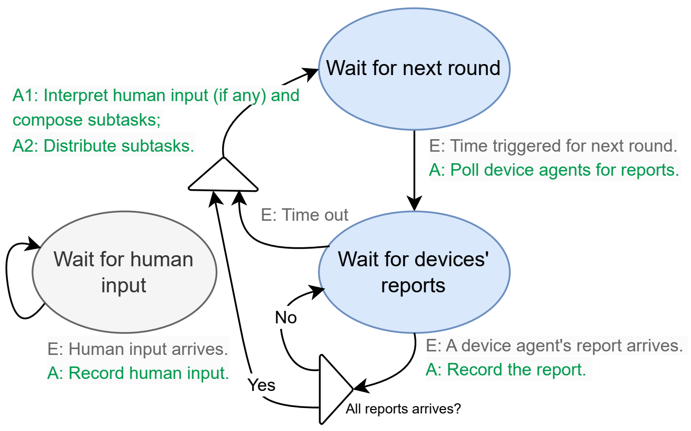

Wait for next round
A1: Interpret human input (if any) and compose subtasks;
A2: Distribute subtasks.
E: Time triggered for next round.
A: Poll device agents for reports.
E: Time out
Wait for human input
Wait for devices' reports
No
E: Human input arrives.
A: Record human input.
Yes
E: A device agent's report arrives.
A: Record the report.
All reports arrives?

Figure 7: State transition diagram of the coordinator.

In the second thread, the coordinator periodically polls device agents for reports. These reports include device status, attributes, subtask completion status, API profile updates, etc. The device status indicates whether the device is operating normally, while the subtask completion status indicates whether the assigned subtasks have been completed, are ongoing, or are not executable.

After the poll, the coordinator waits for the device agents' reports to return. As shown in Fig. 7, when the reports from all polled device agents have been received, the coordinator then interprets the reports, checks if new instructions from humans have arrived, and if necessary, composes and distributes new subtasks for execution by the device agents. Also note from Fig. 7 that a time-out mechanism is installed. This is designed to prevent deadlocks: if some polled device agents fail to return a report, the coordinator can proceed to interpret and act on reports that have been returned. The coordinator concludes a round of operations after assigning all subtasks.

Device Agent Protocol: Fig. 8 shows the state transition diagram of the device agent consisting of four threads of execution. The first thread sends the report to the coordinator upon receiving the coordinator's poll. The second thread pushes a natural language subtask to a queue when the subtask arrives at the agent.

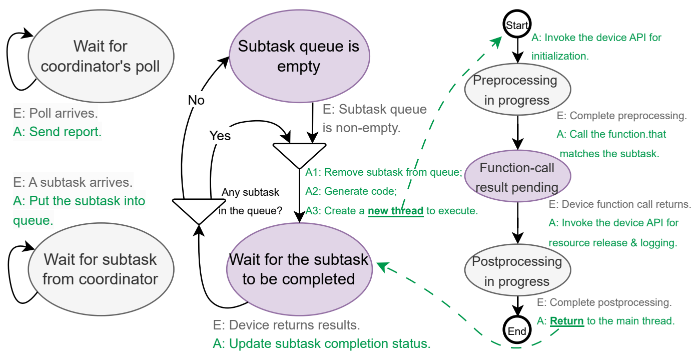

Wait for coordinator's poll
E: Poll arrives.
A: Send report.
E: A subtask arrives.
A: Put the subtask into queue.
Wait for subtask from coordinator
Subtask queue is empty
No
Yes
E: Subtask queue is non-empty.
Any subtask in the queue?
A1: Remove subtask from queue;
A2: Generate code;
A3: Create a new thread to execute.
Wait for the subtask to be completed
E: Device returns results.
A: Update subtask completion status.
Start
A: Invoke the device API for initialization.
Preprocessing in progress
E: Complete preprocessing.
A: Call the function.that matches the subtask.
Function-call result pending
E: Device function call returns.
A: Invoke the device API for resource release & logging.
Postprocessing in progress
E: Complete postprocessing.
A: Return to the main thread.
End

Figure 8: State transition diagram of the device agent.

If the subtask queue is not empty, the agent's third thread pops a subtask from the queue, interprets it in the context of the device API, and generates several lines of executable code (details given in Subsections B and C below).

The fourth thread, responsible for the subtask execution, replaces the pre-defined placeholder in the FSM-based template with the newly generated code. It executes the five-state subtask FSM defined in the code, which instructs the device to perform actions via its API. As illustrated in Fig. 8, the five states of the subtask FSM are: 1) start, 2) pre-processing, 3) subtask function call, 4) post-processing, and 5) end. For preprocessing, the fourth thread invokes the device API function call for initialization and reset, aiming to get the device ready for the upcoming subtask execution. Once the initialization is complete, the fourth thread calls the API function associated with the subtask. When the function call returns, the fourth thread proceeds to post-processing, invoking the device API to release resources and perform logging of essential information. As shown in Fig. 8, upon the device completing the post-processing, the fourth thread returns to the third thread, and then the fourth thread terminates.

After the termination of the fourth thread, the third thread updates the subtask's completion status and checks to see if there are outstanding subtasks in the queue. If so, the third thread repeats the above actions; otherwise, it waits until the queue is filled with a new subtask again before repeating the above actions.

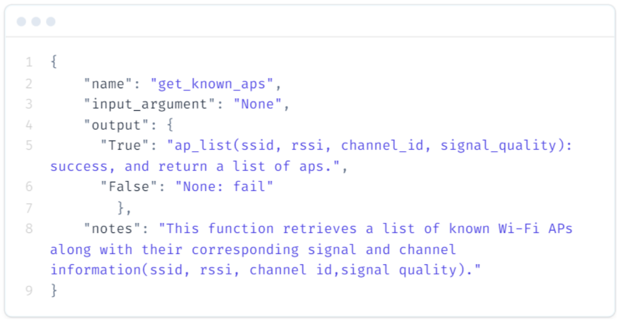
Note that the subtask queue operates with a single-cache mechanism: only the latest subtask is retained in the queue. If a new subtask arrives while another subtask is still in the queue, the new subtask supersedes the one currently in the queue, which is then dropped. The underlying idea is that the absence of execution results for the queued subtask, as observed by the coordinator, implies that the subtask has not been executed. In such cases, the coordinator may generate a new subtask to replace the one in the queue, considering the possibility that the device agent might have failed to execute the subtask on time. This failure could be due to the device agent's slow processing speed, the slow response of the device itself, or other factors. This mechanism is designed to prevent deadlocks and redundant execution of subtasks. In essence, the current protocol operates under a best-effort assumption, where the execution of subtasks by device agents is not guaranteed. Furthermore, even if subtasks are executed, their results may not always be satisfactory. While this paper focuses on a specific protocol, alternative protocols beyond the one explored here are certainly possible.

B. RAG-based Mapping of Subtasks to Device API

An important step the agent needs to complete for the reliable subtask execution is to identify the appropriate API needed for the given subtask. A detailed description of the device's available APIs, such as the user manual or the developer handbook of the device, is typically included in the agent system as an external knowledge base. However, since a developer handbook or user guide may contain the full API description, most of which are irrelevant to the subtask currently assigned to the device and could introduce misleading information, including the whole API handbook as part of the LLM's prompt input is neither efficient nor reliable. Hence, a matching process that precisely retrieves the information of the most subtask-relevant API is necessary.

We developed an RAG module within the device agent to address the mapping problem. The RAG module measures the similarity between the given natural language-based subtask description and the description of each API to find out the most relevant API. Based on the similarity, the RAG module retrieves detailed API information, such as the data type or allowed range of each input argument, for the LLM's later processing.

In our experiment below, detailed information of a device's APIs is structured into a unified JSON format for efficient storage. Fig. 9 gives an example of the JSON format, where the API function "get_known_aps" searches nearby WiFi access points (APs) and returns a list of known APs. No input argument is required for this function call.



```
{
    "name": "get_known_aps",
    "input_argument": "None",
    "output": {
      "True": "ap_list(ssid, rssi, channel_id, signal_quality):
success, and return a list of aps.",
      "False": "None: fail"
        },
    "notes": "This function retrieves a list of known Wi-Fi APs
along with their corresponding signal and channel
information(ssid, rssi, channel id,signal quality)."
}
```

Figure 9: A device API function of JSON format in the external corpus. Note that this function will be used in our experiments in Section IV.

With the JSON-formatted API knowledge base, we now introduce the RAG process with the illustration of Fig. 10. For efficient retrieval, an RAG algorithm requires corpus data chunking as an important pre-processing operation, which segments corpus text into smaller chunks to satisfy the language model's context length limit [10]. Since we are interested in the description of each individual API, a natural treatment is to have each chunk containing the JSON-formatted API description of a single function only. For the next step of data pre-processing, we encode each chunk into a vector embedding using a pre-trained Transformer model (we use SentenceTransformer [11]). Given a subtask specification, the device agent encodes it into another sentence-vector embedding using the same transformer network. The device agent then computes the cosine similarities between the subtask embedding and the API function embeddings in the database, retrieving the API function with the highest score as the best match API for the subtask.

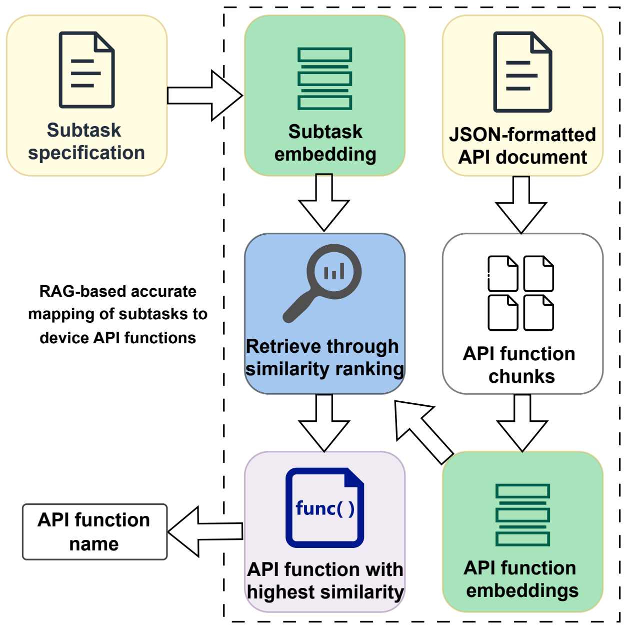

Subtask specification
Subtask embedding
JSON-formatted API document
RAG-based accurate mapping of subtasks to device API functions
Retrieve through similarity ranking
API function chunks
API function name
func( )
API function with highest similarity
API function embeddings

Figure 10: The RAG-based approach involves encoding subtask specifications and device API functions into sentence embeddings for semantic similarity matching.

C. Customized Code Generation with Fine-tuned LLM

With the well-matched API function specification, we now introduce how we fine-tune a Qwen3-1.5B model for reliable and prompt code generation. As discussed in the system design principle (see Section III-B), we want the lightweight LLM to extract necessary input-argument values from the natural language description of the given subtask and compose several lines of executable code according to 1) the API function obtained from the RAG process and 2) the input argument values it just extracted.

Reliability is one of our top concerns for the code generation process. For evaluation, we tested multiple lightweight LLMs (with sizes no larger than 7B) and found through experiments that these models occasionally fail to generate fully functional and executable code. This observation is counterintuitive, especially considering people's general impressions of LLMs' coding ability. With later experiments, we identify the following reasons for this phenomenon: 1) Although significantly more efficient than the code-based counterparts, natural language-based M2M communications naturally introduce randomness to the code generation process. This is due to the potential ambiguity in human language, and 2) lightweight LLMs used in our agent system are less powerful than those cloud-based we are familiar with (and gain impressions from), say GPT-4 and DeepSeek-R1. Lightweight LLMs have limited abilities when the code generation task is complex, e.g., the subtask description contains multiple operations.[1]

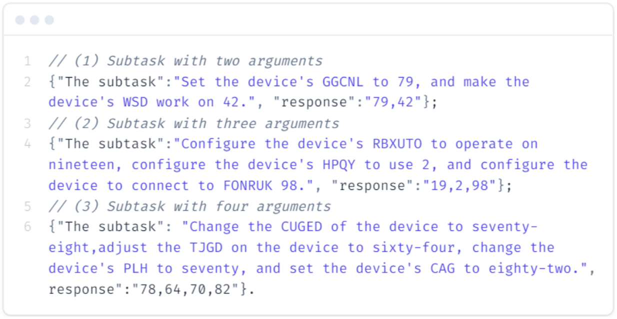
This paper addresses this reliability concern with model adaptation on a task-specific dataset, combined with a two-step problem-solving strategy. Specifically, we fine-tune a Qwen3-1.5B model to precisely extract input argument value(s) from the subtask's natural language description, which serves as the first step of the code generation. In the second step, we already have information about 1) the API function name, which is obtained through the RAG process described in subsection B, and 2) input arguments of the API, which were extracted from the task description through the first step. We ask the LLM to concatenate the information provided (i.e., API function name and associated input arguments) to complete the code generation task.

We now introduce the model fine-tuned for the first-step operation (note: the information concatenation in the second step is simple, and we will not jump into details). Regarding the task-specific data, we craft each data instance based on a triplet of {*ACTION, ATTRIBUTE, VALUE*}, where <ACTION> represents the operation required in a subtask (e.g., set, change), <ATTRIBUTE> refers to the device attribute (e.g., position, temperature), and <VALUE> is the parameter to be set (corresponding to the function input). In the dataset construction process, We randomly generate a triplet instance and use GPT-4 to interpret the triplet into a human language description, and we take the <VALUE> element of the triplet as the associated response . Here is an example:

Triplet: {tweak, ABC, 26}
Human Language Description: ”Tweak the device's ABC parameter to twenty-six.”
Response: ”26.”

Note that the above example represents a simple case with only one input argument. API interfaces requiring multiple input arguments are also commented. To ensure the LLM's generalization ability in the face of possible API inferences, we enrich the diversity of the dataset in terms of input arguments. Specifically, we created subtasks with multiple input arguments by replicating and combining single-argument templates. Fig. 11 illustrates examples with 2, 3, and 4 arguments. In total, 12,000 subtask-argument pairs were generated for each category (1–4 arguments), making the overall size of the dataset 48,000. We randomly choose 10% of the data, which is 4,800 pieces, as the testing set, and the remaining 80% is used for the model fine-tuning process.

[1]While some subtasks look easy and straightforward from a human perspective, requiring a simple action to complete the operation, they may take multiple steps in terms of machine control. For example, heating up a cold meal with a microwave oven looks simple, but it requires several steps of machine command executions: open the door, put the food inside, close the door, turn on the oven, wait a while, turn off the oven, open the door, take out the food, and finally, close the door. When controlling an IoT device with the LLM agent, the code generation process therein must consider every detailed move, making the control task challenging.



```
// (1) Subtask with two arguments
{"The subtask":"Set the device's GGCNL to 79, and make the
device's WSD work on 42.", "response":"79,42"};
// (2) Subtask with three arguments
{"The subtask":"Configure the device's RBXUTO to operate on
nineteen, configure the device's HPQY to use 2, and configure the
device to connect to FONRUK 98.", "response":"19,2,98"};
// (3) Subtask with four arguments
{"The subtask": "Change the CUGED of the device to seventy-
eight,adjust the TJGD on the device to sixty-four, change the
device's PLH to seventy, and set the device's CAG to eighty-two.",
response":"78,64,70,82"}.
```

Figure 11: Examples of generated ”subtask-argument” pair with multiple input arguments.

Finally, LoRA fine-tuning was applied to the agent LLM using this dataset. Regarding the detailed configuration of the LoRA adaptation, we set LoRA_rank = 32, LoRA_alpha = 32, and LoRA_dropout = 0.1. Meanwhile, we have batch_size = 2, learn_rate = 2e-2, and we applied the 8-bit AdamW optimizer in the training process.

IV. Experiments and Case Studies

Subsection A considers the experimental setup introduced in Section II-B, in which an LLMind 1.0 system is tested with the multi-robot warehouse scenario, and demonstrates how the same task could be executed in a more efficient and flexible way with LLMind 2.0. Subsection B presents a real-world case study we have conducted with LLMind 2.0, which serves as a solid validation for the feasibility of the proposed system. Meanwhile, experiments in Subsection B also showcases how distributed agents in LLMind 2.0 facilitate device collaboration in response to external environment changes in an IoT system.

A. *Experiment One: The Warehouse Searching Task*

Recall from Section II that we have assumed a warehouse scenario with N shelves and N robots of various brands. To initialize the vacancy searching task, a warehouse manager gives the following instruction to the robotic system: ”Please check if there are vacant positions on the shelves.” Fig. 3 presents the time of the task execution process and the success rate of code generation with different N being considered in LLMind 1.0. As a comparation, we now give experimental details when LLMind 2.0 is applied to the system with the same user instruction. In the following, we start with the performance of a distributed agent in API function matching and argument extraction. And then, we take a look at the behavior of the overall LLMind 2.0 system (with both the coordinator and distributed agent considered) and benchmark it with LLMind 1.0 in terms of time consumption and the rate of successful task execution.

Before detailed experimental results, we give an implementation overview of the LLMind 2.0 system. In our system, we realized the coordinator LLM with GPT-4 and applied the following parameter setups: temperature = 0.5, top-p =

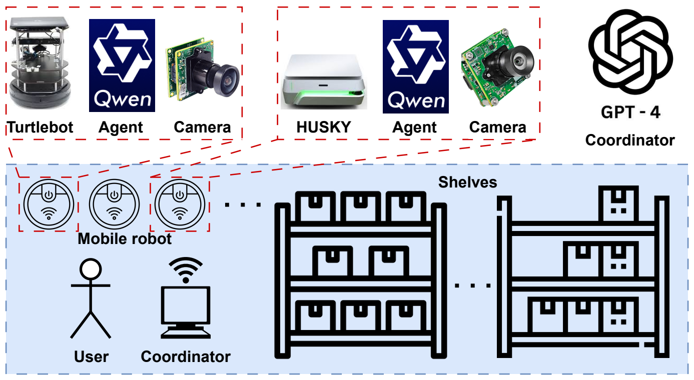
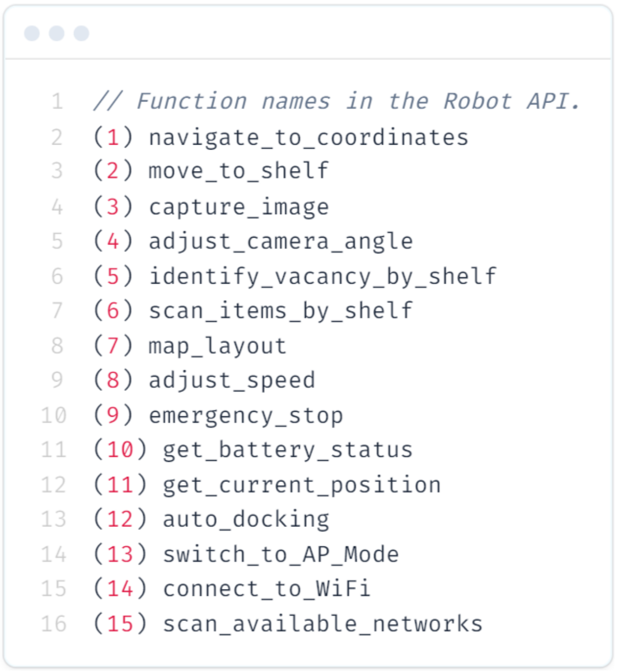


Figure 12: An illustration of the tested warehouse scenario.

1.0, frequency penalty = 0.0. Meanwhile, the distributed agent was fine-tuned based on the LLaMA-2-7B model with the ”subtask-arguments” dataset we built in Section III-C, and it was deployed on a NVIDIA 1060 GPU for each device. For the realization of the similarity measurement, we choose SentenceTransformer [11] as the textual embedding network, and the dimension of the embedding space is set as 768.

We first investigated the two-step procedure for an agent to transform the human-language-based subtask description into executable code: 1) API function mapping, and 2) argument extraction. Experimental results of the two steps are as follows.

For an individual robot, let’s say Robot #1, we assume that the API functions in Fig. 13 are available.



```
// Function names in the Robot API.
(1) navigate_to_coordinates
(2) move_to_shelf
(3) capture_image
(4) adjust_camera_angle
(5) identify_vacancy_by_shelf
(6) scan_items_by_shelf
(7) map_layout
(8) adjust_speed
(9) emergency_stop
(10) get_battery_status
(11) get_current_position
(12) auto_docking
(13) switch_to_AP_Mode
(14) connect_to_WiFi
(15) scan_available_networks
```

Figure 13: API functions that are available to Robot #1. Here we only present the name of each API. For detailed information of the robot’s available API (recorded in JSON format), such as the required input of an API function and the description of the function, we refer readers to Appendix B.

After the coordinator has decomposed the given warehouse searching task into executable subtasks for each robot, the following subtasks were assigned to the robot: 1) Move to shelf one, and 2) Identify the vacancy in shelf one. Fig. 14 presents the API function matching process for the two subtasks. As we can see, API function #2 (i.e., move_to_shelf) and API function #5 (i.e., identify_vacancy_by_shelf) were chosen for realizing Subtasks #1 and #2, respectively. The two selected APIs can well accomplish the required subtask, showcasing the feasibility of the API matching scheme.

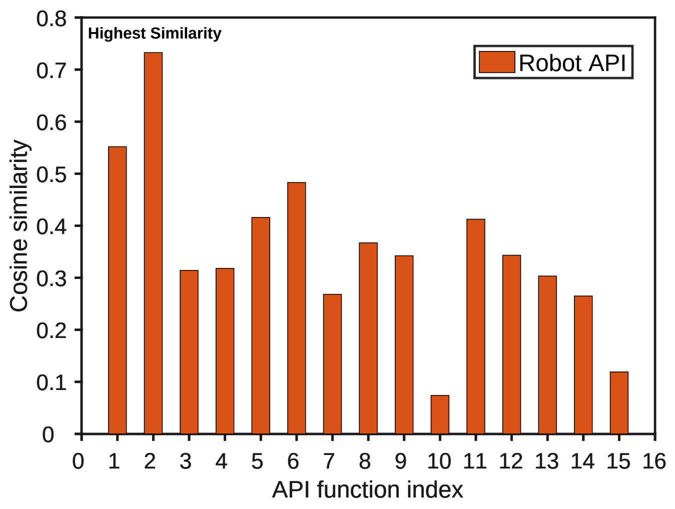
Meanwhile, we also examined the agent’s reliability in the argumentation extraction process. With the selected API functions, we want to enlarge the agent’s success rate in extracting the required arguments for the API function from the human description. Multiple LLMs are tested on for the argument extraction task (see Section III-B for the testing dataset), including 1) powerful cloud-based models, such as Gemini 2.5 Pro, GPT-4o, Chaude-3.7, and 2) Qwen-3 models of different sizes, including the lightweight 0.6B model that serves as our foundation model for the fine-tuning process.

Fig. 15 shows the experimental results of these models on the testing dataset. While those advanced cloud-based models have left us with an impression of being powerful for general tasks, none of them can guarantee 100% reliability in our task. Regarding the Qwen-3 series, the worst performance (i.e., accuracy as low as 37.1%) happens on the smallest 0.6B model. However, even based on the smallest foundational model with the worst performance, the model obtained the best possible performance with no error happening in the testing dataset after fine-tuning, i.e., 100% accuracy in the extraction of API argumentations. This observation highlights the significant performance boost through fine-tuning.

With the RAG-based API matching process and the language model fine-tuned for extracting the API argumentations, an agent can take good care of the code generation task for each individual robot, making it possible the parallel code generation across distributed devices. Thanks to the parallel processing in distributed agents, LLMind 2.0 has a significantly reduced response latency and greatly improved reliability in the code generation process compared with LLMind 1.0. Fig. 16(a) demonstrates LLMind 2.0’s enhanced reliability brought by the distributed code generation. For the LLMind 1.0 setup, the rate of successful code generation decreases when a large N is considered, given that the task is solely undertaken by the central coordinator and can be extremely complex when a huge number of robots get involved. Under the LLMind 2.0 setup, on the other hand, the subtask assigned to a distributed agent remains simple regardless of the number of N, it always considers one robot only. Therefore, no matter how many robots are considered, LLMind 2.0 always shows a very reliable performance in terms of successful code generations for all robots. Meanwhile, as shown in Fig. 16(b), a significant reduction in time consumption is also achieved thanks to the parallel code generation.

B. Experiment Two: Real-world System Demo in WiFi Network

This subsection deploys an LLMind 2.0 system in a real-world WiFi network to validate the feasibility of the proposed



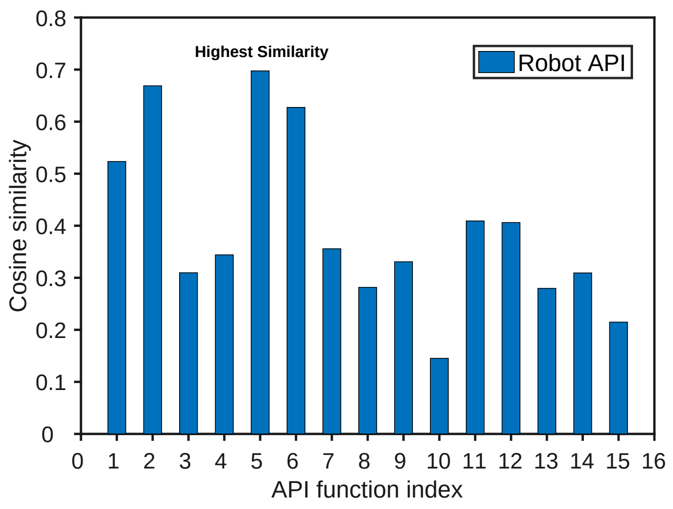


Figure 14: Detailed API matching results of the given warehouse searching task.

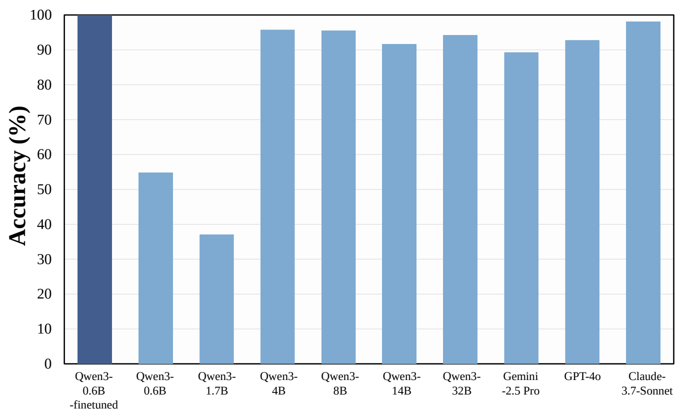


Figure 15: Experimental results showcasing the ability of the different LLMs on the argument extraction task.

system and illustrate key features of the system with two experiments. The first experiment demonstrated the competition between device agents in the LLMind 2.0 ecosystem, while the second experiment highlighted the collaborative behavior among device agents.

Let us start with a quick overview of the system setup.

General System Setup: Fig. 17 illustrates the system setup for the following experiments. There were two WiFi clients connecting to a WiFi access point (AP) with the IEEE 802.11n standard. Client 1 used an FPGA-based software-defined radio (SDR) platform that runs OpenWiFi to realize WiFi functionality; Client 2 had a commercial WiFi adapter. Both clients supported dual-band WiFi functionality, and the AP supported 2.4 GHz and 5 GHz WiFi connections simultaneously. The router had a wired connection to a mini-PC that served as the file server. The two clients kept uploading high-resolution images via TCP to the file server, with each image file having a fixed size of 4MB. Both the clients and the server ran on Linux.

LLMind 2.0 Deployment: To run the LLMind 2.0 system, the two clients were equipped with local device agents connecting to the cloud-based LLMind 2.0 coordinator. Each device agent was supported by an RTX 1060 GPU, a representative setup in terms of the computational ability of IoT devices. The coordinator was built on the cloud with the GPT-4 model. To prevent LLMind 2.0 traffic from influencing the data transmission within the wireless network (such as potential CSMA contentions caused by the LLMind traffic), each agent was connected to the cloud coordinator with a wired network interface card (NIC) other than the wireless NIC described above.

API Functions: The wireless NICs in Client #1 and Client #2 were realized with the OpenWifi project (runs on an FPGA chip) and the RTL8812BU chip, respectively. Both the SDR project and the commercial chip have open-sourced their Linux driver on GitHub, available at [12] and [13], respectively. Detailed introduction about the API functions supported by each NIC is available at the project's GitHub description.

The NIC in Client #1 allows for more flexible operations given the nature of SDR. For example, when CSMA contention happens and backoff operations are required for loser devices, the NIC of a device selects a random number between CW_min and CW_max, and then it assigns the random number as the length of the contention window (CW) of the device [9]. For commercial NICs, the values of CW_min and CW_max are fixed to pre-defined values. The OpenWifi-based NIC, on the other hand, allows for manual configuration of CW_min and CW_max (although it follows the standard setup of CW_min and CW_max by default). This makes the control of this NIC more flexible and user-customized. Fig. 18 shows the JSON-formatted API function description that OpenWifi has developed for the configuration of CW_min and CW_max [12].

On the other hand, the NIC built upon SDR platforms has its disadvantages: its reliability performance, represented by packet error rate (PER), is not as good as a commercial chip under the same channel condition. This is because the SDR project is just for system demonstration purposes. It has neither realized advanced coding/decoding algorithms nor gone through comprehensive testing as a commercial chip may have done for optimized performance. It has also been observed

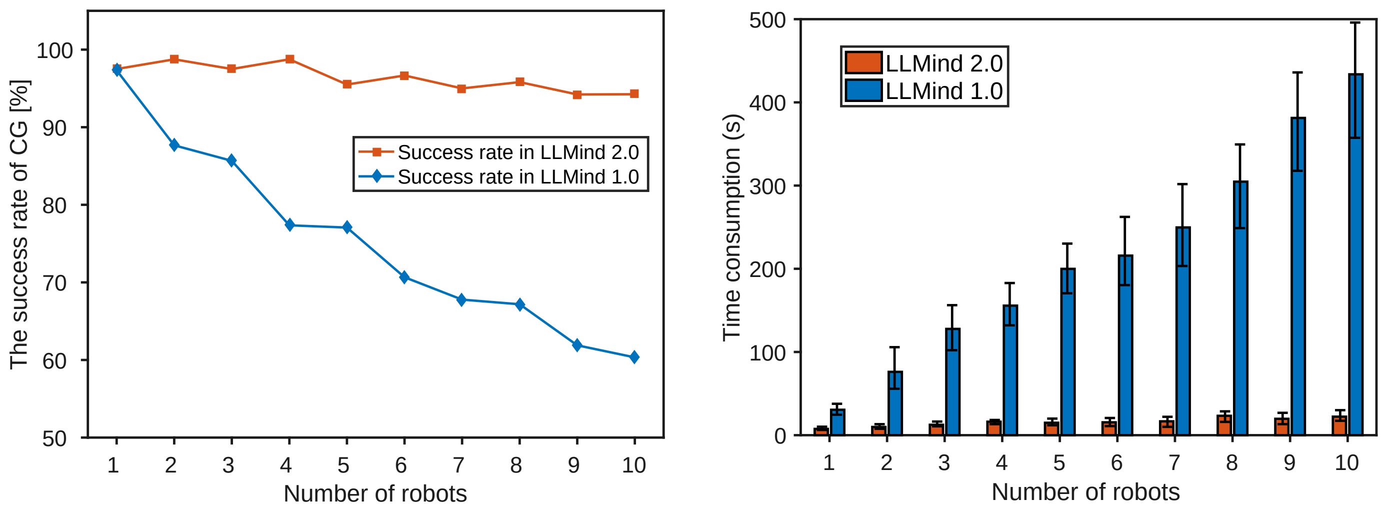


Figure 16: Experimental results showcasing the reduced response latency and improved success rate of the generation process in LLMind 2.0. Here 100 independent tests for the warehouse task are conducted under different N values.

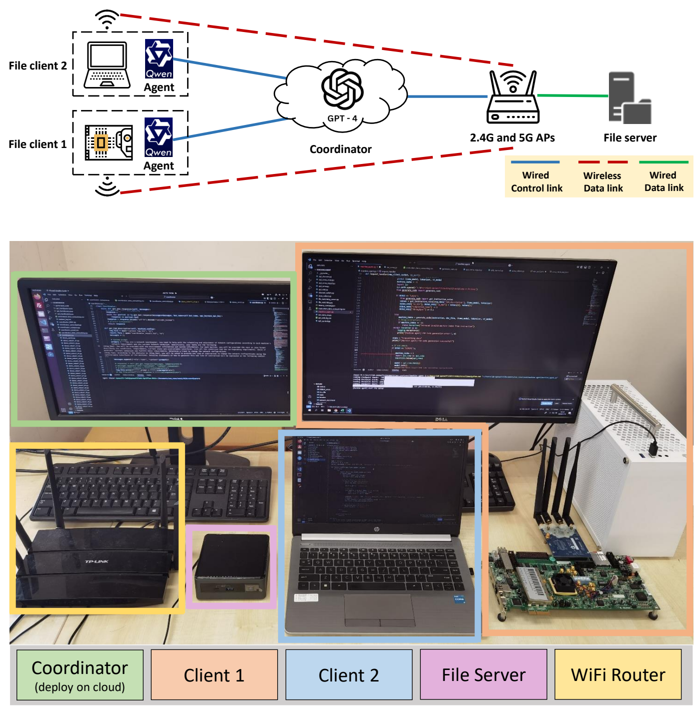
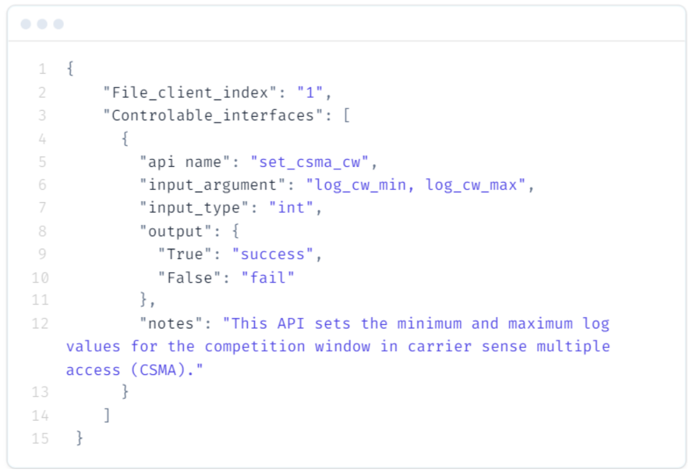


Figure 17: A real-world IoT system developed to showcase the collaborative behavior of the LLMind 2.0 device agent.

that the SDR-based NIC, compared with the commercial NIC, tends to exhibit lower signal transmission power and employs a lower modulation & coding scheme (MCS) index than the commercial NIC.[2]

With the above setup, we now introduce two interesting experiments. **Scenario 1** considered an image uploading task, in which we want the successful uploading of a 4MB image to

[2]According to our analysis on open discussions within the OpenWiFi community, the lower signal transmission is probability due to the absence of power amplifiers in the AD9361 RF extension board, while the low MSC setup are likely caused by the device's intension in maintaining a reasonable PER with its inferior reliability performance.



```
{
    "File_client_index": "1",
    "Controlable_interfaces": [
      {
        "api name": "set_csma_cw",
        "input_argument": "log_cw_min, log_cw_max",
        "input_type": "int",
        "output": {
          "True": "success",
          "False": "fail"
        },
        "notes": "This API sets the minimum and maximum log
values for the competition window in carrier sense multiple
access (CSMA)."
      }
    ]
  }
```

Figure 18: Details about the CW_min and CW_max configuration API for OpenWiFi. Note that the input arguments of the API function are the log values of the CW_min and CW_max.

be accomplished within a pre-defined time threshold to fulfill the user's requirement (here we set the timing requirement as 16 seconds). At the beginning of the experiment, Client 1 was the only device within the network, exclusively occupying all network resources to achieve the desired performance in terms of image uploading time. At that time, the OpenWiFi-based NIC in Client 1 had a default log_CW_min and log_CW_max setup of (10,15).

Shortly thereafter (t_0 in Fig. 19), Client 2 joined the network and began uploading the image. In the competition for transmitting opportunities, Client 2 took around 50% of the network resources (i.e., the CSMA protocol ensured fair transmissions for all devices within the network). However, the OpenWiFi-based NIC in Client 1 required more than half of the transmission chances to fulfill the 16-second timing requirement for the image uploading task, given its low MCS index and high packet retransmission rate. Therefore, as in Fig. 19 (from t_0 to t_1), Client 1 had problems meeting the timing requirement when Client 2 joined the network. Client 2, on the other hand, built the WiFi connection with a more reliable commercial chip and required fewer than half of the transmission chances to meet the 16-second timing requirement (i.e., it had no problem from t_0 to t_1).

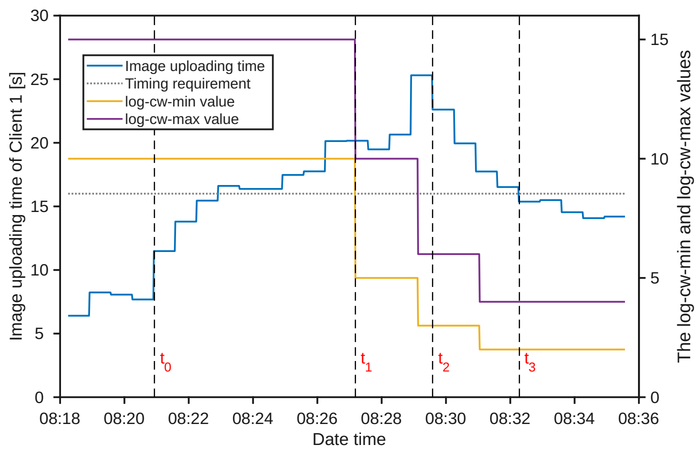


Figure 19: Experimental observation in Scenario 1. Note from this figure that there is a lagging effect in the CW value adjustment of Client 1, which is attributed to the Cubic congestion control algorithm deployed in the Linux TCP stack. The lagging effect found in the experimental result is consistent with the observation reported in [14]. A detailed explanation about this phenomenon is available in Appendix C.

In the execution of the experiment, each device reported its task fulfillment result (i.e., success or failure) and real-time device attributes, including the configuration of (log_CW_min, log_CW_max), packet retransmission rate, MCS index, and the time consumed for the image uploading task, to the coordinator for further decision making. Under the supervision of the system coordinator, Client 1 gradually enhanced its competitiveness of network resources by decreasing the log_CW_min and log_CW_max, eventually reaching the value of (2,4). Consequently, the image uploading time of Client 1 decreased to 15.37 seconds at time t_3, falling below the performance requirement. This improvement is attributed to the EDCF mechanism in WiFi networks [9], where lowering the CW value can reduce Client 1's backoff time in CSMA, thereby giving more packet transmission opportunity to Client 1.

The above experiment illustrated how a device agent could facilitate the competition for system resources under the supervision of the system coordinator. Note that the above competition was a healthy one that enabled more reasonable resource sharing that resulted in better overall system performance, i.e., allowing both devices to meet the 16-second timing requirement. If Client 1 requested so many resources (by further lowering its CW setup) that made the other device fail to meet the timing requirement, the agent in Client 2 would complain to the coordinator and stop Client 1 from doing so.

In general, the competition between devices, which was made possible by the on-device agent, and the supervision of system coordinators, enabled more efficient resource sharing within the LLMind 2.0 system. For more information and a video demonstration of the experiment in Scenario 1, we refer readers to http://123.

Scenario 2: We now consider another scenario where the two devices tried to complete the image transmission task via UDP under strong interference. The interference was caused by a working microwave oven near the testing environment (very close to those devices shown in Fig.17, but not shown within the image). This time, the performance requirement for both clients was keeping the PER not exceeding 20%. At the beginning of the experiment, both Client 1 and Client 2 were connected to the WiFi AP with the 2.4 GHz channel (specifically, Band 7 of the IEEE 802.11n standard) and kept uploading 4M images via UDP to the server as in Scenario 1. After some time (t_0 in Fig. 20), we turned on a microwave oven (we know that microwave ovens typically operate at a frequency band around 2.4 GHz). The following experiment shows how the microwave oven's interference would affect the image transmission of both clients and how the two clients collaborated through information sharing and overcame the strong interference by changing to the 5GHz channel.

Fig. 20 presents the PER throughout the experiment. Before time t_0, the PER values of both clients met the expected performance requirements (i.e., less than 20% PER). Thereafter, when the microwave oven was turned on, both devices suffered from high PER due to the channel interference.

Faced with the challenge, an advantage of Client 1 was that the SDR-based NIC therein allowed the agent to collect more detailed lower-layer networking information than a commercial NIC. That is, the agent of Client 1 could obtain more detailed information about the signal processing process in PHY[3], while the commercial NIC only provided the agent with limited information allowed by the chip manufacturer. Therefore, even if Client 1 had a higher PER (see the blue curve in Fig. 20 before t_0), it provided detailed channel sensing information in its report to the coordinator. Thanks to the channel sensing information provided by Client 1, the coordinator became aware of the wireless interference when the microwave oven started to work. As a result, the coordinator informed Client 2 about the interference and asked it to switch to the 5 GHz channel provided by the AP. Following a disconnection of the 2.4GHz link and a reconnection to the 5GHz link, Client 2's PER value dropped to 6.7% at time t_2, meeting the required PER performance. On the other hand, Client 1 was unable to adjust its channel selection, given that the SDR platform does not allow for channel switching without a system reboot. At time t_3, we turned off the microwave, and the PER of Client 1 returned to normal.

In the above experiment, in which Client 1 shared its sensing information is a "selfless" manner, i.e., helping Client 2 out of trouble even if itself could not get rid of the interference in the 2.4GHz channel. Different from the competition in Scenario 1, the information sharing in Scenario 2 represented the cooperative behaviors between device agents, addressing the external interference with the joint effort of the system.

[3]Some recent works even use the open-sourced SDR project for fine-grained wireless channel sensing [15]–[17]

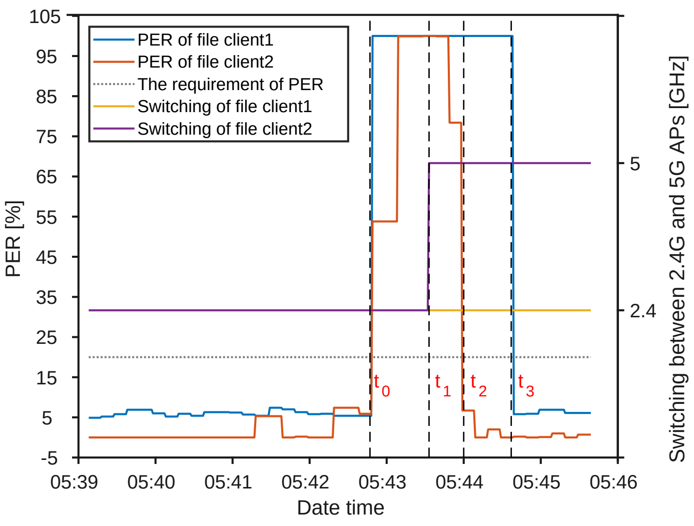


Figure 20: Experimental observation in Scenario 2, in which the UDP protocol is applied. Note that Client 1 used an OpenWiFi board to access the network and cannot switch from the 2.4 GHz AP to the 5 GHz AP without rebooting. Client 2, on the other hand, used a commercial WiFi NIC that allowed channel switching without system rebooting.

For the video demonstration of Scenario 2, we refer readers to http://123.

V. Conclusion and Outlooks

In this paper, we presented LLMind 2.0, a novel distributed IoT automation framework that fundamentally rethinks large-scale IoT device management by harnessing lightweight, embedded LLM-powered device agents and natural language-based machine-to-machine communication. Departing from the traditional centralized approach, LLMind 2.0 offloads device-specific code generation tasks to local agents, which are equipped with fine-tuned LLMs tailored for the control script generation scheme in LLMind 2.0. This distributed architecture not only alleviates the computational bottleneck of a central coordinator but also enables seamless integration and collaboration across heterogeneous devices, regardless of the manufacturer of the device or the programming language supported by the device.

Our design introduces several technical innovations, including a retrieval-augmented generation (RAG) mechanism for precise subtask-to-API mapping and a finite state machine-based code generation pipeline that, when combined with targeted model fine-tuning, significantly enhances both reliability and efficiency. Experimental validation in complex multi-robot warehouse scenarios and real-world WiFi network deployments demonstrated that LLMind 2.0 achieves substantial improvements in scalability, latency, and success rate of task execution, outperforming centralized baselines. Moreover, by localizing sensitive code generation steps, our system provides substantial privacy and security benefits, ensuring that proprietary device data remains on-device.

The results illustrate that natural language can serve as a robust, unifying medium for both human-to-machine and machine-to-machine interactions in IoT, unlocking new levels of flexibility, parallelism, and intuitive collaboration. By open-sourcing our code and fine-tuning datasets, we hope to foster further research into distributed, language-driven automation systems. Looking ahead, we envision expanding LLMind 2.0 to support even larger device ecosystems and more sophisticated collaborative behaviors, further advancing the intelligence and autonomy of future IoT environments.

Notably, our experiments also revealed that distributed device agents can exhibit both competitive and cooperative behaviors in real-world scenarios. In resource-constrained environments, device agents, under the supervision of the system coordinator, efficiently negotiated and adapted to ensure optimal resource allocation that worked best for the overall system in terms of task completion. While in the presence of external interference, they were able to share critical information and assist each other to maintain system performance. This dynamic interplay between competition and collaboration not only enhances system robustness and efficiency but also highlights the potential of distributed LLM-powered agents to autonomously resolve conflicts and support collective problem-solving in heterogeneous IoT ecosystems.

For future work, we will explore scaling LLMind 2.0 to support even larger and more diverse IoT networks, including edge and mobile devices with varying resource constraints. We also plan to investigate more advanced natural language negotiation protocols among device agents to enable richer forms of autonomous collaboration and decision-making. Additionally, extending the framework's privacy and security mechanisms - such as federated learning or on-device model updates - will be crucial as sensitive applications emerge. Finally, we aim to benchmark LLMind 2.0 in more complex, real-world industrial and smart city scenarios to further validate its robustness and adaptability.

Appendix A
Review of LLMind 1.0

This paper builds on our previous work, where we proposed LLMind 1.0 [3], a Language-Code translation method empowered by LLM for M2M communication. In LLMind 1.0, a general-purpose LLM serves as the central coordinator: it first translates human-language instructions into an FSM representation as an intermediate step, then generates device-executable code based on the FSM and distributes it to the respective devices.

In a system with heterogeneous devices, the central coordinator in LLMind 1.0 sequentially translates human-language instructions into executable code for each device through its proprietary API. Specifically, the Language-Code translation pipeline for each device consists of two main steps:

1) **Task Planning:** The central coordinator analyzes the human-language instructions in the context of environmental information and the device's proprietary API, then plans tasks for each device as an FSM. The planning task ensures that the devices work together to follow human-language instructions by performing their individual tasks.
2) **Code Generation:** The central coordinator integrates the proprietary API functions of each device to convert the

FSM output from Step 1 into device-executable code, which is then distributed to the corresponding device.

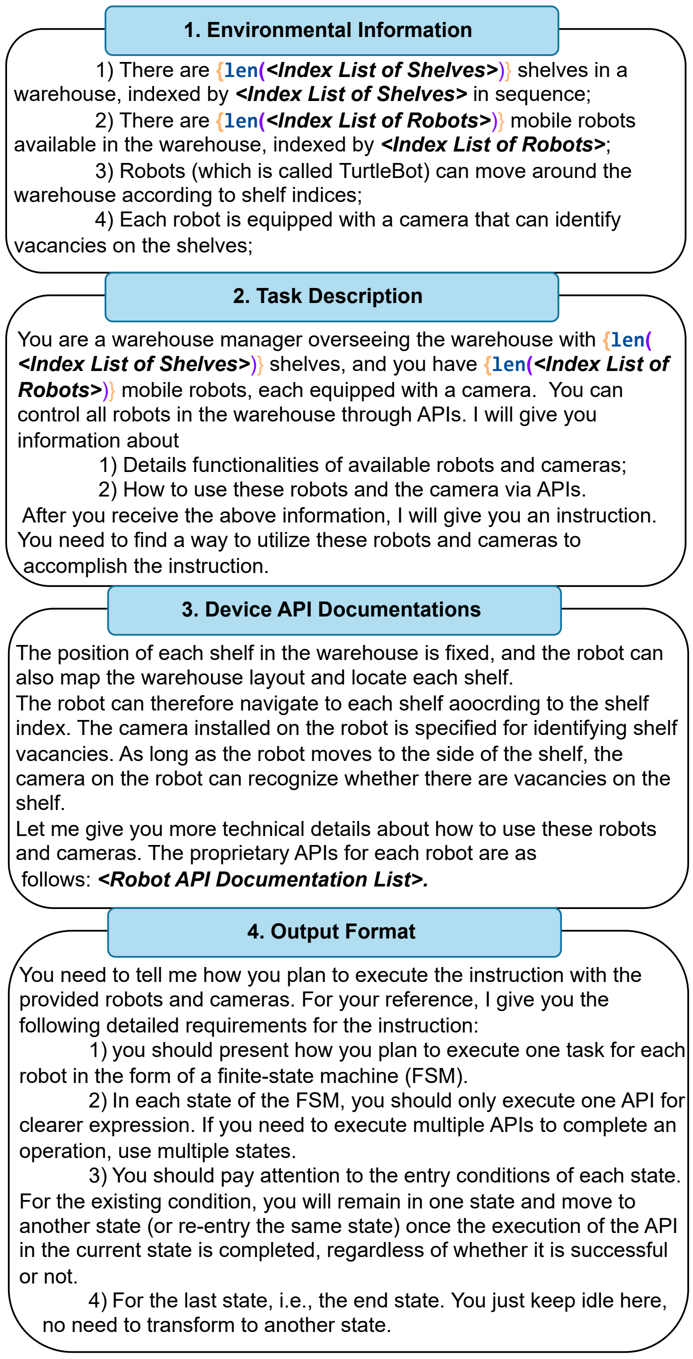
In Step 1 (Task Planning) of the Language-Code translation pipeline, the input prompt provided to the coordinator LLM includes the following components:

1) **Environmental information:** Provides an overview of the scenario, behavioral constraints of the devices, the devices' operating environment, and other factors that may impact task planning.
2) **Task description:** Outlines the coordinator's role in the scenario, the chain of thought of the coordinator, and the objectives for each device's task.
3) **Device API documentation:** Lists API function names, input arguments, outputs, and relevant notes, helping the coordinator LLM understand each device API's functionality.
4) **Output format:** Specifies that the output for each device in the task planning phase should be an FSM that follows the human-language instructions.

In Step 2 (Code Generation) of the Language-Code translation pipeline, the input prompt provided to the coordinator LLM includes the following components:

1) **In-context learning example of FSM implementation:** Includes expert-written state definition and transition implementations for the example FSM, which serves as a reference for programmatically implementing an FSM.
2) **Device API documentation:** Lists API function names, input arguments, and outputs that can be incorporated into executable code.
3) **Output format:** Specifies that the output for each device in the code generation phase should be executable code that corresponds to the FSM output from the task planning phase.

We re-implemented LLMind 1.0 [3] for the multi-robot shelf vacancy search scenario outlined in Section II-A, a typical system with heterogeneous devices. In the task planning phase of the Language-Code translation pipeline, as shown in Part 1 of the prompt in Fig. 21, the environmental information defines a warehouse scenario with N shelves, numbered #1, #2, #3, . . . , # N, and N mobile robots from different brands, also numbered #1, #2, #3, . . . , # N. These robots are capable of navigating to designated shelves within the warehouse. Each robot is equipped with a network camera that can detect vacancies on the shelves. Part 2 in Fig. 21 corresponds to the task description prompt, which specifies that the central coordinator functions as a warehouse manager, analyzes the received human-language instructions, and directs the robots via their APIs to accomplish the received instructions. In Part 3 of Fig. 21, the device API documentations detail the functionalities of the diverse robots. As the N robots are from different brands, their proprietary APIs support different programming languages and are documented in various natural languages. Fig. 22 illustrates 10 sampled API functions with similar functionality from each of the N = 10 robots. These 10 API functions enable a robot to navigate to a designated shelf in the warehouse. Other functionalities of the robot API are further illustrated in Appendix B. In Part 4 of Fig.


1. Environmental Information

1) There are {`len(`***<Index List of Shelves>***`)`} shelves in a warehouse, indexed by ***<Index List of Shelves>*** in sequence;
2) There are {`len(`***<Index List of Robots>***`)`} mobile robots available in the warehouse, indexed by ***<Index List of Robots>***;
3) Robots (which is called TurtleBot) can move around the warehouse according to shelf indices;
4) Each robot is equipped with a camera that can identify vacancies on the shelves;

2. Task Description

You are a warehouse manager overseeing the warehouse with {`len(`***<Index List of Shelves>***`)`} shelves, and you have {`len(`***<Index List of Robots>***`)`} mobile robots, each equipped with a camera. You can control all robots in the warehouse through APIs. I will give you information about

1) Details functionalities of available robots and cameras;
2) How to use these robots and the camera via APIs.

After you receive the above information, I will give you an instruction. You need to find a way to utilize these robots and cameras to accomplish the instruction.

3. Device API Documentations

The position of each shelf in the warehouse is fixed, and the robot can also map the warehouse layout and locate each shelf.
The robot can therefore navigate to each shelf aoocrding to the shelf index. The camera installed on the robot is specified for identifying shelf vacancies. As long as the robot moves to the side of the shelf, the camera on the robot can recognize whether there are vacancies on the shelf.
Let me give you more technical details about how to use these robots and cameras. The proprietary APIs for each robot are as follows: ***<Robot API Documentation List>.***

4. Output Format

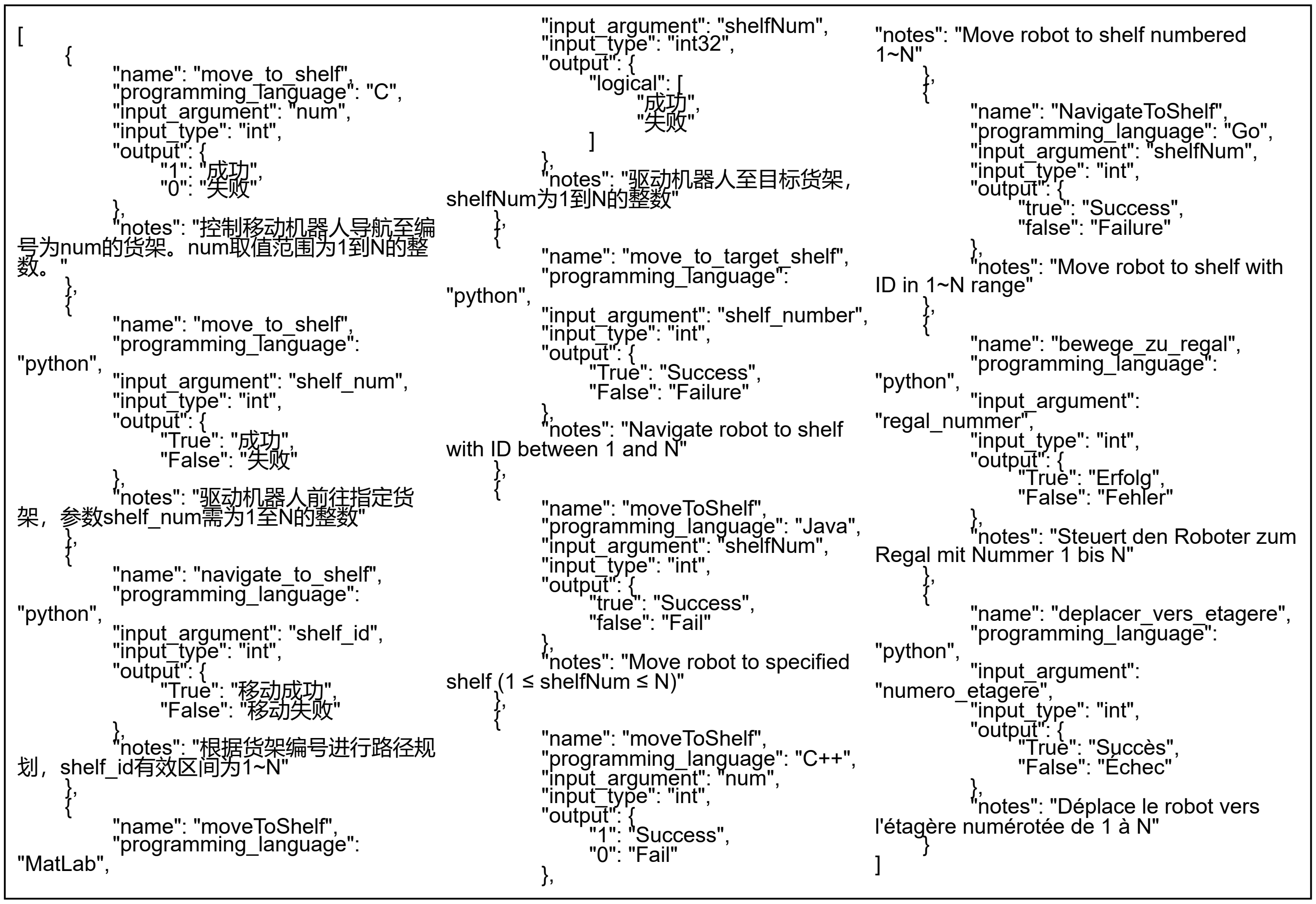
You need to tell me how you plan to execute the instruction with the provided robots and cameras. For your reference, I give you the following detailed requirements for the instruction:

1) you should present how you plan to execute one task for each robot in the form of a finite-state machine (FSM).
2) In each state of the FSM, you should only execute one API for clearer expression. If you need to execute multiple APIs to complete an operation, use multiple states.
3) You should pay attention to the entry conditions of each state. For the existing condition, you will remain in one state and move to another state (or re-entry the same state) once the execution of the API in the current state is completed, regardless of whether it is successful or not.
4) For the last state, i.e., the end state. You just keep idle here, no need to transform to another state.

Figure 21: An illustration of the prompting process during the task planning phase of the Language-Code translation pipeline.

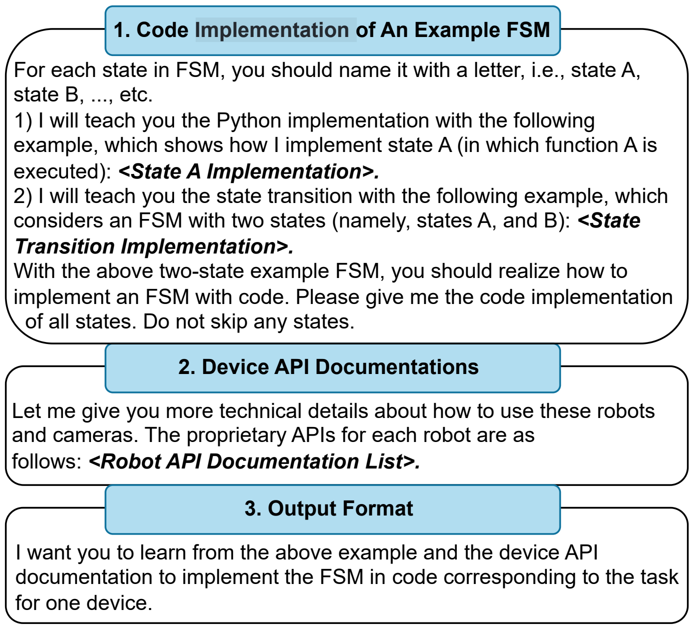
21, regarding the output format, the coordinator must assign a specific task for each robot to execute, ensuring that all robots collaboratively complete the received instructions. The coordinator should represent the assigned task for each robot as an FSM. Each state in the FSM corresponds to a specific robot API function.

In the code generation phase of the Language-Code translation pipeline, Part 1 of the prompt in Fig. 23 provides a Python example of FSM implementation using in-context learning. The FSM consists of two states, A and B, with a transition from A to B. This example enables the coordinator



```
[
    {
        "name": "move_to_shelf",
        "programming_language": "C",
        "input_argument": "num",
        "input_type": "int",
        "output": {
            "1": "成功",
            "0": "失败"
        },
        "notes": "控制移动机器人导航至编
号为num的货架。num取值范围为1到N的整
数。"
    },
    {
        "name": "move_to_shelf",
        "programming_language":
"python",
        "input_argument": "shelf_num",
        "input_type": "int",
        "output": {
            "True": "成功",
            "False": "失败"
        },
        "notes": "驱动机器人前往指定货
架，参数shelf_num需为1至N的整数"
    },
    {
        "name": "navigate_to_shelf",
        "programming_language":
"python",
        "input_argument": "shelf_id",
        "input_type": "int",
        "output": {
            "True": "移动成功",
            "False": "移动失败"
        },
        "notes": "根据货架编号进行路径规
划，shelf_id有效区间为1~N"
    },
    {
        "name": "moveToShelf",
        "programming_language":
"MatLab",
        "input_argument": "shelfNum",
        "input_type": "int32",
        "output": {
            "logical": [
                "成功",
                "失败"
            ]
        },
        "notes": "驱动机器人至目标货架，
shelfNum为1到N的整数"
    },
    {
        "name": "move_to_target_shelf",
        "programming_language":
"python",
        "input_argument": "shelf_number",
        "input_type": "int",
        "output": {
            "True": "Success",
            "False": "Failure"
        },
        "notes": "Navigate robot to shelf
with ID between 1 and N"
    },
    {
        "name": "moveToShelf",
        "programming_language": "Java",
        "input_argument": "shelfNum",
        "input_type": "int",
        "output": {
            "true": "Success",
            "false": "Fail"
        },
        "notes": "Move robot to specified
shelf (1 ≤ shelfNum ≤ N)"
    },
    {
        "name": "moveToShelf",
        "programming_language": "C++",
        "input_argument": "num",
        "input_type": "int",
        "output": {
            "1": "Success",
            "0": "Fail"
        },
        "notes": "Move robot to shelf numbered
1~N"
    },
    {
        "name": "NavigateToShelf",
        "programming_language": "Go",
        "input_argument": "shelfNum",
        "input_type": "int",
        "output": {
            "true": "Success",
            "false": "Failure"
        },
        "notes": "Move robot to shelf with
ID in 1~N range"
    },
    {
        "name": "bewege_zu_regal",
        "programming_language":
"python",
        "input_argument":
"regal_nummer",
        "input_type": "int",
        "output": {
            "True": "Erfolg",
            "False": "Fehler"
        },
        "notes": "Steuert den Roboter zum
Regal mit Nummer 1 bis N"
    },
    {
        "name": "deplacer_vers_etagere",
        "programming_language":
"python",
        "input_argument":
"numero_etagere",
        "input_type": "int",
        "output": {
            "True": "Succès",
            "False": "Échec"
        },
        "notes": "Déplace le robot vers
l'étagère numérotée de 1 à N"
    }
]
```

Figure 22: 10 sampled API functions with similar functionality from 10 diverse robots.



1. Code Implementation of An Example FSM

For each state in FSM, you should name it with a letter, i.e., state A, state B, ..., etc.
1) I will teach you the Python implementation with the following example, which shows how I implement state A (in which function A is executed): ***<State A Implementation>.***
2) I will teach you the state transition with the following example, which considers an FSM with two states (namely, states A, and B): ***<State Transition Implementation>.***
With the above two-state example FSM, you should realize how to implement an FSM with code. Please give me the code implementation of all states. Do not skip any states.

2. Device API Documentations

Let me give you more technical details about how to use these robots and cameras. The proprietary APIs for each robot are as follows: ***<Robot API Documentation List>.***

3. Output Format

I want you to learn from the above example and the device API documentation to implement the FSM in code corresponding to the task for one device.

Figure 23: An illustration of the prompting process during the code generation phase of the Language-Code translation pipeline.

LLM to learn FSM code implementation. The complete code for State A and State Transition implementation enclosed in angle brackets in Part 1 of Fig. 23 is open-sourced at: https://github.com/1155157110/LLMind2.0. The device API documentation in Part 2 of Fig. 23 are the same as that employed in the task planning phase of the Language-Code translation pipeline. With respect to the output format, as illustrated in Part 3 of Fig. 23, the coordinator is responsible for generating executable code based on the FSM output from the task planning phase.

Appendix B
Robot API functionality

The robot API encompasses the intrinsic functionalities of the mobile robot, as well as those of its wireless network module, camera, and custom functions for checking shelf vacancies in the warehouse scenario. Fig. 24 provides an example of a complete robot API implemented in Python. As shown in Fig. 24, the mobile robot provides a comprehensive set of intrinsic API functions, including acquiring battery status, automatic docking, emergency stopping, adjusting movement speed, mapping its surroundings, localization, and navigation based on positional coordinates. Furthermore, the camera provides API functions for adjusting the shooting angle and capturing photos, while the wireless network module offers

API functions for obtaining a list of WiFi hotspots, connecting to networks, and enabling AP mode. In addition, the robot features custom API functions for the warehouse scenario, such as navigating to designated shelves and identifying vacancies or items on those shelves.

Appendix C
Explanation on the lagging effect in Fig.18

We also observed a lagging effect in the image uploading time variation of Client 1, which failed to meet performance requirement when Client 2 joined the network (from t_0 to t_1 in Fig. 18), and then began to decrease after LLMind 2.0 lowered the log_CW_min and log_CW_max values (from t_1 to t_2 in Fig. 18). This phenomenon is mainly attributed to the fact that the image uploading time is measured at the application layer and is inevitably influenced by TCP behavior. The Cubic congestion control algorithm, deployed in the Linux TCP stack, attempts to maintain a relatively constant TCP window size to improve network stability [14]. This leads to longer network convergence times during TCP connection initialization or topology changes [14]. For example, testbed experiments reported in [14] show that the network convergence time during initialization can be approximately 200–300 seconds.

In scenario 1, from t_0 to t_1 in Fig. 18, when Client 2 joined the network and occupied 50% of the network resources, the transmission delay of Client 1's packets increased, and Client 1's TCP window size was gradually reduced with each round of 4MB image uploads. Consequently, the image uploading time of Client 1 gradually increased during this period. From t_1 to t_2, after the log_CW_min and log_CW_max values for Client 1 were lowered, Client 1 aggressively increased its preemption of network resources, leading to intensified contention with Client 2 over a short time. As a result, Client 1's TCP window size temporarily continued to shrink to cope with network degradation, causing a further rise in the image uploading time. From t_2 to t_3, the image uploading time of Client 1 gradually decreased and eventually met the performance requirement as the log_CW_min and log_CW_max values were reduced.

References

[1] V. Ashish, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, p. I, 2017.

[2] Z. Deng, W. Ma, Q.-L. Han, W. Zhou, X. Zhu, S. Wen, and Y. Xiang, "Exploring deepseek: A survey on advances, applications, challenges and future directions," *IEEE/CAA Journal of Automatica Sinica*, vol. 12, no. 5, pp. 872–893, 2025.

[3] H. Cui, Y. Du, Q. Yang, Y. Shao, and S. C. Liew, "Llmind: Orchestrating ai and iot with llm for complex task execution," *IEEE Communications Magazine*, 2024.

[4] B. Xiao, B. Kantarci, J. Kang, D. Niyato, and M. Guizani, "Efficient prompting for llm-based generative internet of things," *IEEE Internet of Things Journal*, 2024.

[5] İ. Kök, O. Demirci, and S. Özdemir, "When iot meet llms: Applications and challenges," in *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 2024, pp. 7075–7084.

[6] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian *et al.*, "Toolllm: Facilitating large language models to master 16000+ real-world apis," *arXiv preprint arXiv:2307.16789*, 2023.

[7] W. Chen, Z. Li, and M. Ma, "Octopus: On-device language model for function calling of software apis," *arXiv preprint arXiv:2404.01549*, 2024.

[8] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis, 2023," *URL https://arxiv.org/abs/2305.15334*, 2023.

[9] P. B. Y. C. Chang, "Ieee 802.11 dcf," *IEEE TRANSACTIONS ON MOBILE COMPUTING*, vol. 4, no. 4, 2005.

[10] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, no. 1, 2023.

[11] "Sentencetransformer," https://sbert.net/docs/package_reference/sentence_transformer/SentenceTransformer.html.

[12] "openwifi," https://github.com/open-sdr/openwifi.

[13] "Rtl8812bu," https://github.com/fastoe/RTL8812BU.

[14] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[15] H. B. Thameur and I. Dayoub, "Sdr implementation of a real-time testbed for spectrum sensing under mimo time-selective channels for cognitive radio applications," *IEEE Sensors Letters*, vol. 5, no. 8, pp. 1–4, 2021.

[16] Y. Xu, R. K. Amineh, Z. Dong, F. Li, K. Kirton, and M. Kohler, "Software defined radio-based wireless sensing system," *Sensors*, vol. 22, no. 17, p. 6455, 2022.

[17] X. Wei, L. Zhang, W. Yuan, F. Liu, S. Li, and Z. Wei, "Sdr system design and implementation on delay-doppler communications and sensing," in *2023 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2023, pp. 1–6.

```
[   {
        "name": "navigate_to_coordinates",
        "input_argument": "x, y",
        "input_type": "float, float",
        "output": {
            "True": "success",
            "False": "fail"
        },
        "notes": "This function navigates the robot
to a specific point in the warehouse, given by the x
and y coordinates."
    },
    {
        "name": "move_to_shelf",
        "input_argument": "num",
        "input_type": "int",
        "output": {
            "True": "success",
            "False": "fail"
        },
        "notes": "This function navigates the mobile
robot according to the shelf number. The setting
range of num is an integer from 1 to N."
    },
    {
        "name": "capture_image",
        "input_argument": "None",
        "input_type": "null",
        "output": {
            "True": "success",
            "False": "fail"
        },
        "notes": "This function captures a instant
image from the robot's camera."
    },
    {
        "name": "adjust_camera_angle",
        "input_argument": "angle",
        "input_type": "float",
        "output": {
            "True": "success",
            "False": "fail"
        },
        "notes": "This function adjusts the camera's
angle to a specified value, helping the robot to better
scan areas or shelves."
    },
    {
        "name":
"identify_vacancy_by_shelf",
        "input_argument": "shelf_id",
        "input_type": "int",
        "output": {
            "True": "vacancy found",
            "False": "no vacancy"
        },
        "notes": "This function identifies whether
there are vacancies on a shelf numbered by shelf_id,
The range of shelf_id is an integer from 1 to N."
    },
    {
        "name": "scan_items_by_shelf",
        "input_argument": "shelf_id",
        "input_type": "int",
        "output": {
            "True": "items found",
            "False": "no items"
        },
        "notes": "This function scans a specific shelf
(identified by 'shelf_id') to detect any items placed on
it. The 'shelf_id' is an integer from 1 to N."
    },
    {
        "name": "map_layout",
        "input_argument": "none",
        "input_type": "null",
        "output": {
            "True": "map generated",
            "False": "mapping failed"
        },
        "notes": "This function generates a map of
the room layout, including aisles and potential
obstacles."
    },
    {
        "name": "adjust_speed",
        "input_argument": "speed",
        "input_type": "float",
        "output": {
            "True": "success",
            "False": "fail"
        },
        "notes": "This function adjusts the robot's
speed to a specified value, allowing it to move faster
or slower."
    },
    {
        "name": "emergency_stop",
        "input_argument": "none",
        "input_type": "null",
        "output": {
            "True": "robot stopped",
            "False": "stop failed"
        },
        "notes": "This function immediately stops
the robot's movement, used in case of an emergency
or safety hazard."
    },
    {
        "name": "get_battery_status",
        "input_argument": "none",
        "input_type": "null",
        "output": {
            "True": {
                "msg": "battery power status
retrieved",
                "status": 0
            },
            "False": "battery status retrieval failed"
        },
        "notes": "This function returns information
about the battery's charge level."
    },
    {
        "name": "get_current_position",
        "input_argument": "none",
        "input_type": "null",
        "output": {
            "True": {
                "msg": "position retrieved",
                "x": 0,
                "y": 0
            },
            "False": "position retrieval failed"
        },
        "notes": "This function retrieves the current
position of the robot."
    },
    {
        "name": "auto_docking",
        "input_argument": "none",
        "input_type": "null",
        "output": {
            "True": "docking successful",
            "False": "docking failed"
        },
        "notes": "This function allows the robot to
autonomously dock itself to a charging station or
designated dock."
    },
    {
        "name": "switch_to_AP_Mode",
        "input_argument": "none",
        "input_type": "null",
        "output": {
            "True": "mode switch successful",
            "False": "mode switch failed"
        },
        "notes": "This function switches the robot's
network mode to Access Point (AP) mode, allowing
for direct communication with other robots."
    },
    {
        "name": "connect_to_WiFi",
        "input_argument": "ssid, password",
        "input_type": "string, string",
        "output": {
            "True": "WiFi connected",
            "False": "connection failed"
        },
        "notes": "This function connects the robot to
a WiFi network using the provided credentials (SSID
and password)."
    },
    {
        "name": "scan_available_networks",
        "input_argument": "none",
        "input_type": "null",
        "output": {
            "True": {
                "msg": "networks scanned",
                "ssid_list": [
                    "aaa",
                    "bbb"
                ]
            },
            "False": "scanning failed"
        },
        "notes": "This function scans for all
available WiFi networks in the vicinity and returns a
list of network names (SSIDs) for the user to choose
from."
    }
]
```

Figure 24: A complete robot API implemented in Python with 15 API functions.