IoTMigrator: LLM-driven Embedded IoT Code Migration across Different OSes for Cloud-device Integration

Yingqi Peng[1], Kaijie Gong[1], Yi Gao[1]*, Hao Wang[1], Wei Dong[1]†
[1]Zhejiang University, Hangzhou, China
{yingqipeng, gongkj, gaoyi, haowang, dongw}@zju.edu.cn

Abstract

The increasing prevalence of embedded systems has necessitated manufacturers to migrate product code, transferring existing products to new embedded operating systems (OSes) for getting better compatibility and performance. Since manufacturers' product code predominantly employs the Thing Specification Language (TSL) paradigm for cloud connectivity, migrated code consequently adheres to the same TSL standard. However, embedded code migration under the TSL paradigm proves more complex than conventional code migration. Neither outline-based code generation nor common code translation techniques can adequately address this challenge, despite their prevalence in existing systems. There exists a growing demand for a algorithm tailored to TSL paradigm embedded code migration. In response to this demand, we have developed IoTMigrator that employs a multi-agent pipeline to handle the issue. The key insight of our algorithm is the TSL enhancer, specifically designed for the characteristics of the TSL paradigm, which serves as a crucial component in the agent pipeline. To demonstrate the superiority of our algorithm, we have established our own benchmark, which includes six tasks across two OSes, RIOT and Zephyr. We adopted two key metrics: compilation pass rate and task completeness score. The experiment results show that our algorithm outperforms the baseline by an average of at least 50.5% for pass rate and 13.0% for completeness across all tasks in RIOT, and at least 83.4% for pass rate and 18.4% for completeness in Zephyr. This work will be open-sourced in the future.

1 Introduction

To allow embedded devices to connect to cloud platforms, cloud platforms have introduced Thing Specification Language (TSL) paradigm (Cloud, 2024). There are now no fewer than 10 types of TSL models, such as Alibaba Cloud TSL (Cloud, 2023a), Tencent Cloud TSL (Cloud, 2023c), Huawei Cloud TSL (Cloud, 2023b), and so on. These TSLs include detailed specifications of device properties, events, services, etc. Device manufacturers must program according to the specific content of the different cloud platforms' TSL, enabling their devices to access the cloud platform.

To implement the various functions of TSL, programmers need to possess a certain understanding of TSL and a certain level of embedded OSes programming. Now, a growing number of manufacturers need to be compatible with other new embedded OSes to keep up with the trend. For example, Midea (Huawei, 2025) announced its support for Harmony OS to embrace the Harmony OS ecosystem. Thus the challenge of how to efficiently enable embedded devices to implement newly emerging OSes is a meaningful issue.

The breakthroughs of LLMs (OpenAI, 2023a), which have not only captured the attention of both academic and industrial communities but also revolutionized cross-OS code migration. Traditional code generation methods, such as outline-based code generation (Doe and Smith, 2025; Le et al., 2024) that first generate an outline and then produce code based on the outline, as well as code translation algorithms (Yang et al., 2024) that translate algorithmic problems into languages like C++ or Java, have both demonstrated outstanding performance. However, traditional code generation based on outline or code translation with LLMs cannot address code migration for embedded OSes well. As shown in Fig. 1, the outline-based approach used the wrong libraries. Since the libraries <Adafruit_SSD1306.h> and <WiFi.h> were used in the reference code, leading the LLM to mistakenly believe that the code for RIOT-OS should also use the corresponding libraries like <ssd1306.h> and <net/emcute.h>. To better prove our claim, we

* Corresponding author.
† Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 19245–19257

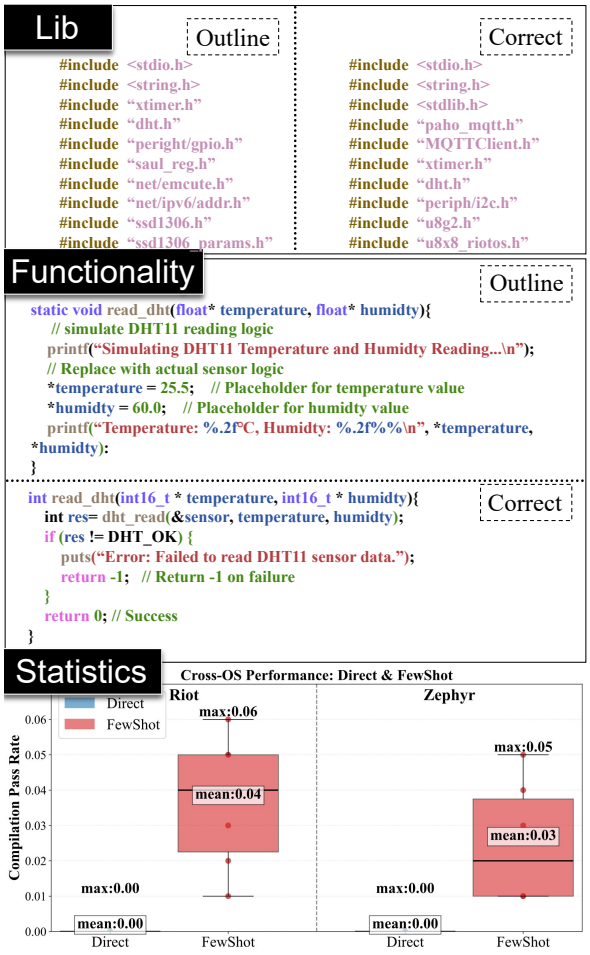

Figure 1: An illustrative example is that LLMs cannot handle code migration well. LLMs frequently import incorrect libraries and may substitute functional code with comments during embedded code generation. Our preliminary experiments demonstrates that both the Direct and FewShot based on the outline fail to generate the embedded codes.

show the statistics result with the GPT-4o, which includes direct method and few shot method across RIOT and Zephyr OSes. Among them, the few shot method utilizes the communication templates we provided to reduce programming complexity. The accuracy rates of both methods are notably low, with the direct approach's rate being virtually zero (we consider accuracy rates below 1% as zero). Moreover, the task completeness of LLM-generated code is insufficient as LLM often uses place holders and simulation data (as shown in Fig. 1 Functionality, traditional methods don't generation the true code to read data from DHT11 but generation the simulation temperature and humidity data). On the other hand, the code translation is typically applied to algorithm programming where the output varies based on different inputs and can be debugged through variable tracing. In contrast, embedded programming frequently operates without conventional inputs since it primarily reads sensor data rather than inputs provided by users. Consequently, the conventional code translation methods are not suit for the embedded programming well.

Confronted with these challenges, recent work explores how to better leverage LLMs for code migration or generation. Some scholars have proposed multi-agent collaborative framework (Shen et al., 2025), tool-using code generation (Zhang et al., 2024), and so on. Nevertheless, none of these methods can effectively accomplish the task of code migration.

This is mainly due to the following reasons: ①. **The misleading influence of reference code provided by users.** Embedded OSes often have more low-level code while the reference code tends to be higher-level, such as Arduino. Errors may ensue if an LLM is affected by the reference code as LLM uses reference code's libraries during migration. Library errors can lead to a series of subsequent function errors and ultimately compilation failure. Moreover, some operations required by Arduino may not be necessary in others, for example, some embedded OS libraries such as RIOT OS don't require connecting to WiFi in the code while Arduino does. ②. **Task completion may be compromised during code generation.** For embedded code, it generally has weaker encapsulation than high-level languages, making it significantly more challenging for LLM to generate successfully compiled code. Consequently, LLM needs an iterative process to ensure the code can compile successfully. However, during the iterative process, the LLM may progressively simplify the code by stripping out functionality to overcome compilation challenges. Although this approach may eventually yield successfully compiled code, it often comes at the expense of significantly decreasing functional completeness.

To address these issues, we propose for the first time an embedded code migration strategy, named **IoTMigrator**. We design a multi-agent pipeline specifically tailored for TSL's unique characteristics. The multi-agent pipeline consists of three agents: outline generator, TSL enhancer and code generator. The three agents each serve distinct functions: outline generator mitigates the impact of the reference code, TSL enhancer enhances task functional completeness by processing tasks according

to TSL's unique characteristics, code generator generates the target code.

Given the current absence of open-source benchmarks for embedded systems, particularly those employing the TSL paradigm, we build a corpus comprising six typical products both in RIOT and Zephyr OSes. Additionally, our experimental design evaluates compilation pass rate and task completeness score. Our algorithm demonstrates significant improvements, achieving an average increase of 138.9% in compilation pass rate in RIOT and 342.7% in Zephyr. For task completeness score, it shows consistent enhancements with average improvements of 26.7% in RIOT and 27.6% in Zephyr.

2 Related Work

2.1 Technology without LLMs

Traditional non-LLM code generation techniques can be used to generate templates, configuration files, models, etc. Code generation technologies include template engines (Ronacher, 2010; Foundation, 2003), code generators (Team, 2012; Swagger, 2015), meta programming (Lombok, 2009; Team, 2005), and DSLs (Parr, 1989; Foundation, 2008). However, template engine technologies cannot adapt to the code for embedded OSes. Code generators can only generate client SDKs and server stubs based on the OpenAPI specification. Metaprogramming technologies cannot generate complete code. DSLs are only used for parsing to generate parsers and lexical analyzers. Additionally, code migration technology is also unreliable, as it is primarily designed for non-embedded OSes code.

2.2 Technology based on LLMs

Typical LLMs include the GPT series (OpenAI, 2023b), BERT (Devlin et al., 2018), and T5 (Raffel et al., 2020). At present, tools and platforms that combine LLMs for code generation or migration include GitHub Copilot (OpenAI, 2021), Amazon CodeWhisperer (Services, 2022), ChatGPT (OpenAI, 2023a), and DeepSeek (DeepSeek, 2025). In practice, these LLMs are unable to directly handle code migration for embedded OSes, with their compilation pass rates close to 0% (shown in statistics part of Fig. 1).

The most advanced researches, such as two-stage code generation(Liu et al., 2023) and fine tuned method (Doe et al., 2024) , they requires a large corpus but collecting corpus for embedded development is significantly more challenging than other common programming languages. As for methods without fine tuning like CodeAgent (Zhang et al., 2024) and AutoIoT (Shen et al., 2025) are not specifically designed for embedded code generation with TSL paradigm. As generalized models, they exhibit limited performance during generation.

3 Design

We propose a system comprising three collaborative agents, as shown in Fig. 2.

The three-agent pipeline operates as follows: (1) Outline Generator: it not only performs user code analysis to extract key information and code logic, but also generates sub-tasks. We record key information as metadata which includes key parameters (e.g. pin parameters, communication parameters) and key functions (e.g. PWM function and ADC function). In addition, it extracts the reference code portion corresponding to the sub-tasks and summarizes the code logic. It ultimately generates a outline containing extracted metadata and sub-tasks. (2) TSL Enhancer: it processes sub-tasks following the outline, which exploits the integration of a rating mechanism and a tool chain. TSL can be fundamentally divided into two components: communication and peripheral control. Moreover, these two components are independent and can be generated separately. More importantly, generating these two components separately is less challenging than direct generation, thereby reducing the likelihood of the LLM compromising code functionality to ensure compilation success. Therefore, this agent automatically generates communication and peripheral control code separately based on the outline content. To ensure the function completeness and data quality, it exploits the rating mechanism to evaluate the generated code while utilizing tool chains to perform retrieval and compilation testing. Finally, the codes for both communication and peripheral control components have undergone multiple rounds of compilation testings, resulting in high quality without compromising completeness. And TSL enhancer takes these two codes with explanatory prompt as final output. (3) Code generator: leveraging the communication and peripheral control codes from TSL enhancer, it generates successfully compiled and high completeness code through utilizing tool chains including retrieving, compilation testing and library detection.

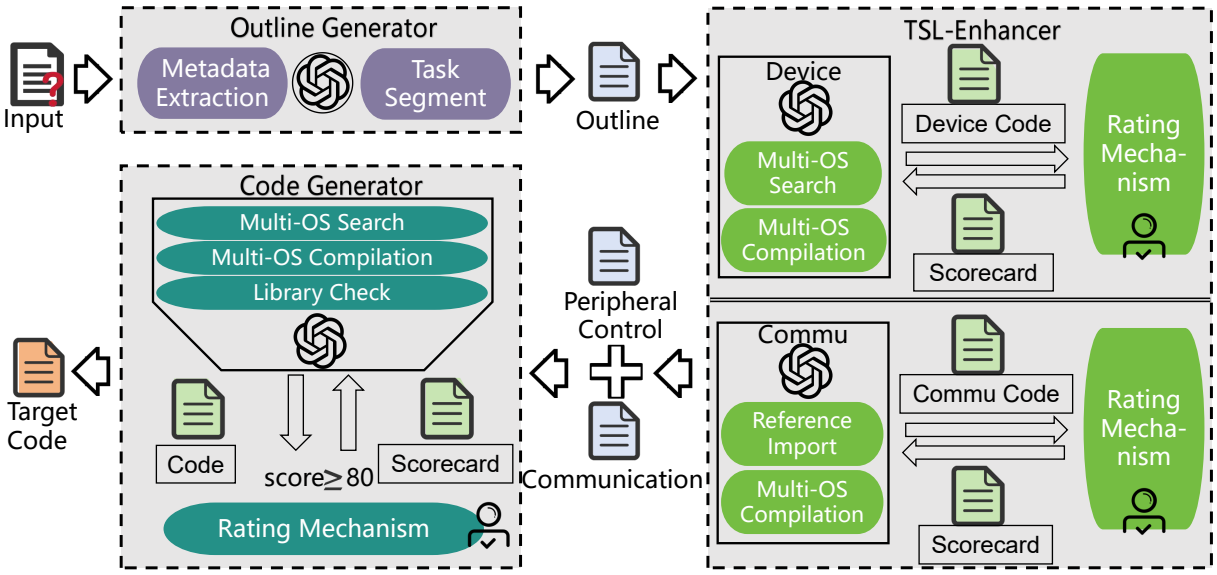

Figure 2: The overview of IoTMigrator. IoTMigrator is an multi-agent pipeline composed of three specialized agents: an Outline Generator, a TSL Enhancer, and a Code Generator. These agents can leverage various tools and rating mechanisms to accomplish their tasks.

The subsequent sections provides technical explanations of the paradigm how three agents work and what the rating mechanism, tool chains are.

3.1 Outline Generator

The objective of IoTMigrator is to maximize the generated code's evaluation score ϑ under fixed input conditions. The fixed inputs are the user's requirements r (users can freely choose whether to include TSL) and the reference code c. The score ϑ is mainly based on whether the code can compile successfully. A score greater than 0 is assigned only if the code compiles successfully; otherwise, the score remains 0. However, we also conduct manual check. If the generated code has obvious content deficiencies, it will only be counted as 0.5 (e.g., only completing the peripheral control part). If the deficiencies are particularly severe, it will be counted as 0 (e.g., containing only the Main function).

$$\max \vartheta \quad \text{subject to} \quad r \& c \tag{1}$$

The reference code generally has inclusion of essential parameters such as MQTT connection parameters and peripheral pin definitions, as well as reference libraries and functions (e.g., PWM and ADC) necessary for task fulfillment. Meanwhile, the reference code incorporates complete TSL parameters, allowing users to optionally provide TSL parameters or not. Furthermore, the reference code may implied logic for operations like re-connection and data parsing. To migrate the code better, outline generator needs to extract the valid information called metadata from the reference code. In parallel, outline generator segments user's requests into multiple sub-tasks while the extracted metadata is aligned to the corresponding sub-task. Outline generator outputs an outline $\mathfrak{O}$ which lists all the sub-tasks with each sub-task containing its corresponding metadata.

3.2 TSL Enhancer

The multi-task generator agent receives the outline $\mathfrak{O}$ as input and generates communication code $\mathbb{M}$ and peripheral control code $\mathbb{P}$ as outputs.

Given that TSL tasks typically consist of both communication and peripheral control components and this two components are completely decoupled, TSL enhancer independently handles the code generation for these two components. The agent separately generates communication code $\mathbb{M}$ and peripheral control code $\mathbb{P}$ based on the sub tasks from the outline $\mathfrak{O}$. During peripheral control code generation, TSL enhancer processes all device-related sub-tasks with performing compilation testing and scoring on the generated code. For the code generation of the communication module, the same process as that of the peripheral control module is followed. The compilation testing and scoring mentioned in these two code generation processes are performed by specialized tools from our tool chain and rating mechanisms. Our own dedicated

tool chain $\mathcal{T}$ comprises: multi-OS search $\mathcal{T}_{search}$, multi-OS compilation $\mathcal{T}_{compile}$, template import $\mathcal{T}_{template}$. In addition, we develop the rating mechanism $\mathcal{R}$ which employs a rating agent to score so that TSL enhancer can improve the task completeness via comments from rating mechanism.

Due to the niche nature of embedded systems, direct web searches often yield irrelevant or low-quality results. Therefore, our approach primarily references examples and tests from code repositories and we extract all repository examples and tests into a structured CSV file. When the TSL enhancer activates $\mathcal{T}_{search}$, it generate the key words according the query, then search agent is activated and retrieves the most relevant examples/tests from the CSV. Furthermore, existing tools (e.g., LangChain) primarily support compilation for common programming languages, necessitating manual adaptation for embedded systems. We configure and encapsulate the compilation environments for RIOT OS and Zephyr into our tool chain, enabling automated compilation testing by the agent. Finally, we develop a template import tool that covers common communication protocols in both RIOT and Zephyr. TSL enhancer dynamically selects relevant communication template, then TSL enhancer generates the codes that satisfy the sub-tasks of communication through learning the template.

The workflow of TSL generating communication and peripheral control code leveraging the toolchain and the rating system is shown in Alg.1.

3.3 Code Generator

Code generator employs the communication and peripheral codes as inputs and ultimately generates the code required by users as the target code. For the purpose of the successful-compiled code, code generator utilizes a tool chain that incorporates $\mathcal{T}_{search}$, $\mathcal{T}_{compile}$ and library check $\mathcal{T}_{check}$ to assist in the process.

The workflow of the code generator is similar to that of the TSL enhancer, differing only in the tool chain employed. The tool chain's capabilities of $\mathcal{T}_{search}$ and $\mathcal{T}_{compile}$ retain identical to those in previous tool chains, except for $\mathcal{T}_{check}$. $\mathcal{T}_{check}$ is responsible for library missing issues by analyzing the compilation logs and it is automatically run after $\mathcal{T}_{compile}$ without selection from the agent. Upon detecting missing libraries, it employs a transform model (Sentence-Transformers, 2020) to encode the missing library names, which are subsequently matched against encoded library names of the repository through cosine similarity. The agent considers the library potentially existing but with a different name if similarity exceeds the predefined threshold. Otherwise, the agent considers the libraries is indeed absent and prompts the code generator not to use them.

Algorithm 1 TSL_Enhancer Workflow

Input: Maximum iterations T, score threshold ς
Output: Generated code or $\varnothing$

```
t ← 1, initialize prompt;
while t ≤ T do
    Select tool ∈ {T_compile, T_search, T_template};
    if tool = T_compile then
        code ← Generate code;
        log ← T_compile(code);
        if ϑ(code) > 0 then
            // compilation passed
            (score, comment) ← R(code);
            if score ≥ ς then
                return code;   // Success case
            else
                prompt+=comment;
            end
        else
            prompt+=log;
        end
    else
        (output) ← T_search ∨ T_template;
        // For search/template
        prompt+=output;
    end
    t ← t + 1
end
return the last compiled code if code else ∅;
// Failure case
```

4 Experiments

4.1 Setup

The system used in my experiment is Linux Ubuntu 22.04, with the software being VSCode 1.92.2, Python version 3.10.9, OpenAI version 0.28.0, and Conda version 23.1.0.

Benchmark.

The six representative applications we introduce utilizes TSL's property, event and service. These six applications comprehensively cover the four fundamental categories of IoT applications: environment sensing, device actuation, user iteration and security alerting from Xiaomi's Miot spec.

We assume that the user needs to migrate common Arduino code into uncommon OS code such as RIOT-OS and Zephyr. Therefore, we use the Arduino code as the reference code. Each of our applications implemented in Zephyr and RIOT-OS has clearly defined TSL settings.

1) **Thermometer**: It can not only sense temperature but also detect humidity and upload the data to the cloud, belonging to environment-aware category with TSL={property: temperature, humidity}. 2)**Doorbell**: It is essentially a doorbell that can upload press notification to the cloud, belonging to user iteration with TSL={event: BellEvent}. 3)**Switch**: It is a dual-row switch that uses two buttons to control two LED lights, belonging to device actuation with TSL={property: LeftButton, RightButton, LeftState, RightState}. 4)**Smart light**: It uses two buttons to independently control three colors and three brightness levels of the LED, belonging to device actuation category with TSL={property: Brightness, Color}. 5)**Smoke detector**: It monitors air condition and upload to the cloud as well as triggers air pollution alerting, belonging to environmental sensing category and security alerting with TSL={property: AirCondition; Event: AirAlarm}, which triggers an alarm event when the air quality detected is below a threshold. 6)**Display**: It can display temperature and humidity on an LED screen, trigger an high temperature alerting, and also accept cloud-based control to turn the display on or off. It belongs to environmental sensing and security alerting with TSL={property: temperature, humidity; event: HighTemperatureAlarm; service: ControlDisplay}.

Baselines.

We compare IoTMigrator with three other algorithms by implementing them on our own. Although some of them have public code repositories, they can't be directly adapted to our experiments. Therefore, we implement them according their design and provide them the same prompts as us. 1) **RagDebug.** It has a tool chain that combines multi-OS search and multi-OS compilation tools to generate the code. However, it uses the tools in a fixed sequence of first conducting a multi-OS search and then continuously using the multi-OS compilation. The algorithm of IoTMigrator, CodeAgent and AutoIoT are different from it, because only this one follows a fixed sequence while the others choose the tools freely by LLM. 2) **CodeAgent.** It leverages a tool chain that is different from us to generate the target code by code generator, however, it doesn't equipped with outline generator and TSL enhancer. Its tool chain is almost the same as ours except that it has web search tool and api search tool more than us while it lacks a library check tool. 3) **AutoIoT.** It consists of three components: design generation, code generation, code improvement. The most different part between us and its is that it doesn't compile each code segment but compile the whole code. Furthermore, it doesn't have rating mechanism during the whole process but have a code improvement component. Last but no least, it doesn't have library check tool since this tool is designed by ourselves.

Table 1: Ablation Results Comparison (Compilation Pass Rate & Task Completeness Score)

OS	Algorithm	Ther	Bell	Switch	Light	Detect	Display
Compilation Pass Rate							
RIOT	-Enhance	0.88	0.50	0.50	0.63	0.38	0.38
	-Rating	0.83	0.83	0.86	0.57	0.71	0.63
Zephyr	-Enhance	0.81	0.40	0.38	0.33	0.06	0.06
	-Rating	0.94	0.25	0.67	0.50	0.25	0.83
Task Completeness Score							
RIOT	-Enhance	4.62	3.75	4.12	3.88	3.75	4.50
	-Rating	4.50	4.33	3.86	3.25	3.86	4.22
Zephyr	-Enhance	4.67	4.00	3.86	3.83	3.88	4.00
	-Rating	4.83	4.00	3.67	4.33	4.00	4.00

Note: -Enhance is Ablation-Enhance, -Rating is Ablation-Rating.

Table 2: Experimental Results of DeepSeek (Compilation Success Rate & Task Completion Score)

Algorithm	Compilation Pass Rate						Task Completeness Score					
	Ther	Bell	Switch	Light	Detect	Display	Ther	Bell	Switch	Light	Detect	Display
IoTMigrator	**0.90**	**0.80**	**0.83**	**0.83**	**0.89**	**0.83**	**4.75**	**4.60**	**4.83**	**4.67**	**4.89**	**4.75**
AutoIoT	0.50	0.33	0.50	0.33	0.50	0.13	4.50	4.00	4.25	3.50	3.80	3.50
CodeAgent	0.08	0.75	0.75	0.21	0.15	0.04	4.54	3.43	4.50	3.00	4.08	4.55
RagDebug	0.83	0.33	0.50	0.17	0.07	0.09	3.00	3.83	2.50	2.67	4.00	4.45

Metrics. We compare IoTMigrator and several baselines by measuring the following evaluation metrics. *Compilation pass rate*: The fraction of correct compilation out of all compilation. *Task Completeness Score*: This is the score selected by GPT-4o from {1, 2, 3, 4, 5} based on the scoring criteria we provided, to evaluate the task completeness of all the generated code.

4.2 Compilation Pass Rate

As shown in Figure. 3, across both RIOT and Zephyr OSes, IoTMigrator demonstrates significant performance advantages: in RIOT OS it outperforms AutoIoT by 50.5%, CodeAgent by 147.3%, and RagDebug by 219.1% on average. While in Zephyr these performance gaps widen sub-

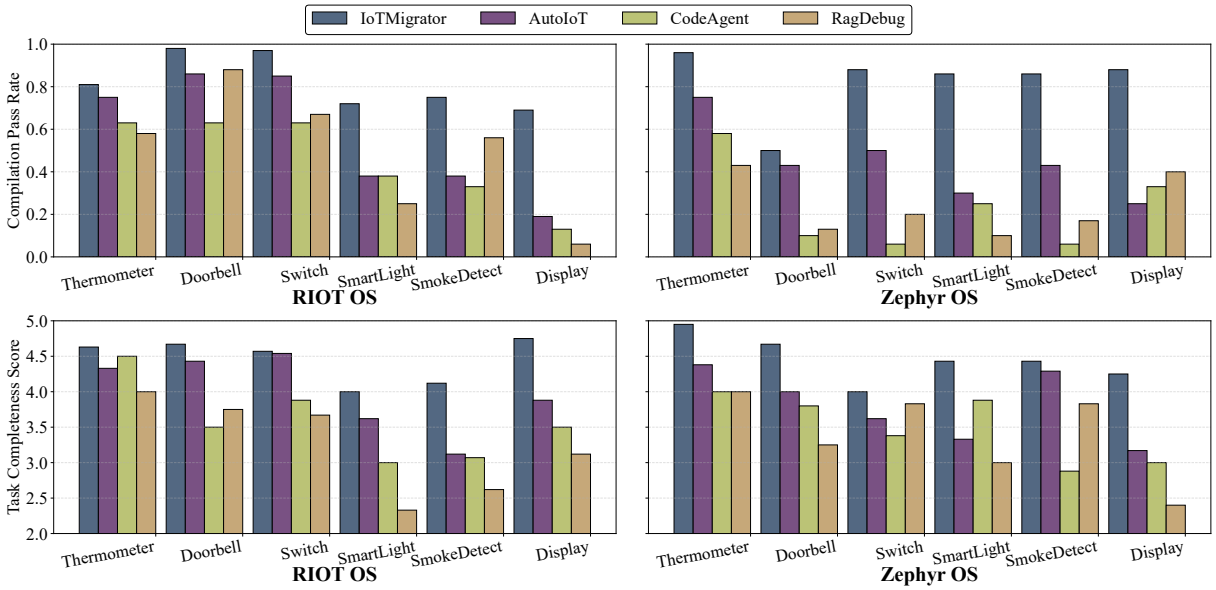

Figure 3: The compilation pass rate and completeness rate of different algorithms in RIOT and Zephyr OSes.

stantially since IoTMigrator surpassing AutoIoT by 83.4%, react by 428.2%, and RagDebug by 516.4% on average. Besides, IoTMigrator has imporved its task completeness score by 13.0%, 26.2%, and 41.0% respectively compared to AutoIoT, CodeAgent and RagDebug in RIOT. Similar for Zephyr, IoTMigrator has also improved its task completeness score by 18.4%, 29.1%, and 35.4% respectively compared to AutoIoT, CodeAgent and RagDebug in Zephyr. According to the experimental results on compilation pass rate, IoTMigrator performs the best, while AutoIoT outperforms CodeAgent and RagDebug. Although CodeAgent shows slightly better performance than RagDebug, both algorithms exhibit unsatisfactory results. In terms of task completeness score, the overall experimental findings also follow the trend: Iotmigrator > AutoIoT > CodeAgent > RagDebug. We believe the reasons for such performance are as follows:

(1) Even with the support of tool chains, poorly designed algorithms still fail to deliver satisfactory performance. Both CodeAgent and RagDebug are the algorithms employ the tool chain, with their main difference lying in their tool chain selection strategies. However, experimental results show that although CodeAgent generally outperforms RagDebug, neither achieves satisfactory performance. The primary reason is that although both algorithms leverage tool chains, their overall architectural designs lack refinement. They generate code in a monolithic manner, without modular testing or improvement m, resulting in unsatisfactory task completeness score and compilation pass rate.

(2) Unlike CodeAgent and RagDebug, AutoIoT adopts a more sophisticated design by decomposing the process into multiple components. After the code generation component completes its task, AutoIoT further refines the generated code in code improvement component. Additionally, it performs compilation tests after each improvement iteration. However, its code generation still follows a monolithic approach rather than employing modular generation and testing like TSL enhancer of IoTMigrator. In contrast, our IoTMigrator incorporates a dedicated TSL Enhancer specifically designed for the TSL paradigm. It decomposes TSL tasks into two parts, generates corresponding outline sub-tasks, and conducts compilation tests for each part. Finally, our code generator significantly improves compilation pass rates and mitigates potential degradation in task completeness. Because code generator only needs to integrate the TSL enhancer's output with tool chain and a rating mechanism.

4.3 Ablation Experiment

We designed two ablation algorithms: Ablation-Enhance, Ablation-Rating. The first one is the IoTMigrator variant with the TSL Enhancer removed. And the other is the IoTMigrator version excluding the rating mechanism. Table. 1 shows the effects demonstrated by our model after undergoing ablation experiments. In RIOT OS, IoTMigrator demonstrates significant improvements in compila-

Table 3: Performance Comparison Across Benchmarks (Recall Scores)

OS	Algorithm	Benchmarks												Mean Recall	
		Thermometer		Doorbell		Switch		Smart Light		Smoke Detect		Display		API	Lib
		API	Lib	API	Lib	API	Lib	API	Lib	API	Lib	API	Lib		
RIOT	IoTMigrator	**0.74**	**0.88**	**0.99**	**1.00**	**0.98**	**1.00**	**1.00**	**0.92**	**0.95**	**0.88**	**0.98**	**0.98**	**0.94**	**0.96**
	AutoIoT	0.64	0.83	0.77	0.90	0.92	0.92	0.67	0.66	0.86	0.72	0.70	0.67	0.76	0.78
	CodeAgent	0.60	0.83	0.60	0.67	0.89	0.88	0.80	0.69	0.61	0.60	0.65	0.54	0.69	0.70
	RagDebug	0.50	0.74	0.84	0.92	0.86	0.89	0.59	0.46	0.82	0.75	0.57	0.42	0.70	0.70
Zephyr	IoTMigrator	**0.81**	**0.83**	**0.85**	**0.75**	0.66	**0.75**	**0.77**	**0.82**	**0.87**	**0.84**	**0.89**	**0.78**	**0.81**	**0.80**
	AutoIoT	0.72	0.67	0.69	0.50	0.65	0.65	0.31	0.54	0.71	0.71	0.57	0.52	0.61	0.60
	CodeAgent	0.56	0.53	0.76	0.37	0.59	0.46	0.65	0.66	0.49	0.57	0.67	0.58	0.62	0.53
	RagDebug	0.57	0.64	0.73	0.62	**0.75**	0.67	0.55	0.54	0.67	0.57	0.30	0.30	0.60	0.56

Note: The numerical values presented in the table represent recall rates, including both API recall and library (Lib) recall metrics. Besides, we have highlighted the maximum values in bold for intuitive visualization.

tion pass rate compared to Ablation-Enhance and Ablation-Rating by 63.4% and 14.3% respectively. The performance gains are even more pronounced in Zephyr OS, with enhancements reaching 464.3% and 66.1% against the same baselines. Similarly, for task completeness score, the system achieves performance boosts of 9.0% and 11.9% in RIOT OS, while maintaining consistent improvements of 10.4% and 7.9% in Zephyr OS over Ablation-Enhance and Ablation-Rating respectively.

The experimental results demonstrate that the TSL enhancer is critically important for IoTMigrator's performance, directly determining both compilation pass rates and task completeness scores. In contrast, the rating mechanism exhibits significantly weaker impact on compilation rates compared to the TSL Enhancer, but its contribution to task completion metrics is comparable to that of the TSL enhancer.

4.4 The Impact of LLMs

To ensure the generalization ability of our algorithm across different LLMs, we conducted additional experiments using DeepSeek-V3, with the results detailed in Tab.2. Our algorithm demonstrates significant performance improvements across all benchmarks: For compilation pass rate, 433.29% average improvement over RagDebug, 634.32% average improvement over CodeAgent, 176.07% average improvement over AutoIoT. For task completeness score, 45.92% average improvement over RagDebug, 21.00% average improvement over CodeAgent, 22.01% average improvement over AutoIoT. These results demonstrate strong evidence that our algorithm exhibits excellent generalization capabilities across different models.

4.5 API&Lib Analysis

To comprehensively analyze all the baselines, We calculated the average recall rates for both APIs and libraries across all benchmarks for the four algorithms, as detailed in Tab. 3. The experimental results demonstrate that IoTMigrator achieves the highest performance metrics across both RIOT and Zephyr platforms. The significantly higher recall rates of IoTMigrator compared to the other three algorithms demonstrate its superior capability in correctly utilizing required libraries and functions, which directly contributes to its higher compilation pass rate. Furthermore, the more functions it employs, the more comprehensive task it implements, thereby explaining its exceptional task completenes performance. This provides a micro-level explanation for IoTMigrator's outstanding overall performance.

4.6 Cross-OS adaptability

The prompts used in our approach are designed to be adaptable to various operating systems. They are intentionally kept generic, as they focus on extracting code logic and application parameters rather than being tied to specific APl calls, libraries, or syntax. These core elements-logic and parameters-are fundamental across programming languages and are not exclusive to Arduino. As a result, the method is not inherently restricted to Arduino-based inputs. To demonstrate the generality of the method, we have conducted a experiment: migrating RlOT-based code to Zephyr-based code.

Table 4: Compilation pass rate of migrating RlOT-based code to Zephyr-based code.

System	Thermometer	Doorbell	Switch	SmartLight	SmokeDetect	Display
IoTMigrator	0.88	0.38	0.87	0.94	0.62	0.88
AutoIoT	0.83	0.14	0.29	0.06	0.43	0.43
CodeAgent	0.50	0.06	0.19	0.06	0.06	0.45
RagDebug	0.75	0.20	0.33	0.25	0.08	0.33

Table 5: Task Completeness score of migrating RIOT-based code to Zephyr-based code.

System	Thermometer	Doorbell	Switch	SmartLight	SmokeDetect	Display
IoTMigrator	5.00	4.38	4.62	4.86	4.75	4.50
AutoIoT	4.67	4.00	4.43	4.29	4.14	3.57
CodeAgent	4.50	3.14	3.62	3.88	4.00	3.80
RagDebug	4.00	2.80	3.50	3.75	3.80	3.50

Tab. 4 and Tab. 5 show that our method significantly outperforms the other baselines, demonstrating generality and avoiding overfitting to Arduino.

4.7 Case Study

To validate the reliability of the completeness score assigned by our LLM judge, we invited 7 experts in the IoT domain to perform a human evaluation. Seven human experts evaluated the same set of 18 code samples. These samples were systematically selected by extracting 3 implementations from each of the six tasks (6 tasks × 3 samples = 18 total). The table below presents our complete evaluation data.

Table 6: Results of the Human Evaluation

ID	Task	Sample	Human Avg	GPT Score
1	Thermometer	1	4.0000	5.0000
2	Thermometer	2	2.8571	3.0000
3	Thermometer	3	3.7143	4.0000
4	Doorbell	1	4.4286	5.0000
5	Doorbell	2	3.2857	3.0000
6	Doorbell	3	3.5714	3.0000
7	Switch	1	3.8571	4.0000
8	Switch	2	4.0000	3.3300
9	Switch	3	4.4286	5.0000
10	SmokeDetect	1	2.2857	2.0000
11	SmokeDetect	2	3.1429	4.0000
12	SmokeDetect	3	4.4286	5.0000
13	SmartLight	1	3.4286	3.0000
14	SmartLight	2	3.2857	4.0000
15	SmartLight	3	3.2857	3.3300
16	Display	1	2.8571	2.3300
17	Display	2	3.5714	3.0000
18	Display	3	4.5000	4.6700

Subsequent mathematical analysis revealed a strong Pearson correlation ($r \approx 0.849, p < 0.01$) between LLM and expert ratings, demonstrating that GPT's scoring trends closely align with human consensus averages. Notably, this high correlation persists even when accounting for low inter-rater reliability among human experts (ICC ≈ 0.090), suggesting the LLM effectively emulates aggregated human judgment.

5 Conclusion

For the embedded code migration problem using the TSL paradigm, we have developed IoTMigrator, a customized multi-agent pipeline solution specifically designed to leverage TSL's unique characteristics to address this challenge. Our framework comprises three specialized agents: The Outline Generator extracts critical information and decomposes the task into subtasks; The TSL Enhancer leverages TSL paradigm characteristics to separately handle communication and peripheral control subtasks; The Code Generator synthesizes the TSL enhancer's outputs into final executable code. In the last, we addressed the corpus scarcity challenge by constructing a dedicated benchmark, through which we evaluated performance using both compilation pass rates and task completeness score. Furthermore, we conducted ablation studies, generalization tests and fine-grained API/library analysis

6 Limitations

To address the migration issues for embedded OS, we have made substantial efforts and achieved certain results. However, our IoTMigrator still faces the following two issues:

- IoTMigrator currently cannot address custom driver issues. We will continue to refine it in the future.
- The current experiments are still limited, and further testing can be conducted on new OS and new TSL tasks in future work.

Despite these existing limitations in IoTMigrator, this work can still contribute to embedded TSL paradigm code migration and enhance the efficiency of embedded code migration.

7 Acknowledge

This work was supported in part by the National Natural Science Foundation of China under Grant 62272407, in part by the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under Grant 2023C01033.

References

Alibaba Cloud. 2023a. Alibaba cloud iot platform tsl documentation.

Alibaba Cloud. 2024. Thing model concepts and usage limits. Official documentation on the definition, functional types (properties, services, events), and usage limits of the Thing Model.

Huawei Cloud. 2023b. Huawei cloud iot platform tsl documentation.

Tencent Cloud. 2023c. Tencent cloud iot hub tsl documentation.

DeepSeek. 2025. Deepseek-v3: An open-source ai model compatible with openai api. https://cloud.tencent.com/developer/article/2486237. Accessed: 2025-01-16.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

John Doe and Jane Smith. 2025. Mapcoder: Multi-agent code generation for competitive problem solving. In *International Conference on Artificial Intelligence*, pages 123–132, New York, USA. AI Society, ACM.

John Doe, Jane Smith, and Alice Johnson. 2024. Knowledge graph based explainable question retrieval for programming tasks. In *Proceedings of the 2024 International Conference on Software Engineering (ICSE)*, pages 1234–1245. IEEE.

Apache Software Foundation. 2003. Apache freemarker: A template engine for java. https://freemarker.apache.org/. Accessed: 2023-10-10.

Eclipse Foundation. 2008. Xtext: A framework for developing programming languages and domain-specific languages. https://www.eclipse.org/Xtext/. Accessed: 2023-10-10.

Huawei. 2025. Harmonyos connect: A new chapter for midea's smart home appliances. Case study on how Midea Group integrates with HarmonyOS Connect to enable smart home appliance interoperability and enhance user experience.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *Proceedings of the International Conference on Learning Representations (ICLR)*, New York, NY, USA. ICLR, OpenReview.net.

Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. Codegen4libs: A two-stage approach for library-oriented code generation. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*, pages 471–483. IEEE.

Project Lombok. 2009. Lombok: Java library to reduce boilerplate code. https://projectlombok.org/. Accessed: 2023-10-10.

OpenAI. 2021. Github copilot: Your ai pair programmer. https://copilot.github.com/. Accessed: 2023-10-01.

OpenAI. 2023a. Gpt-4 technical report. https://cdn.openai.com/papers/gpt-4.pdf. Accessed: 2025-01-16.

OpenAI. 2023b. Openai api documentation. https://platform.openai.com/docs. Accessed: 2025-01-16.

Terence Parr. 1989. Antlr: A powerful parser generator for reading, processing, and translating structured text or binary files. https://www.antlr.org/. Accessed: 2023-10-10.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Armin Ronacher. 2010. Jinja2: A modern and designer-friendly templating language for python. https://jinja.palletsprojects.com/. Accessed: 2023-10-10.

Sentence-Transformers. 2020. all-minilm-l6-v2.

Amazon Web Services. 2022. Amazon codewhisperer: Ai-powered code generator. https://aws.amazon.com/codewhisperer/. Accessed: 2023-10-01.

Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. Autoiot: Llm-driven automated natural language programming for aiot applications. *arXiv preprint arXiv:2503.05346*.

Swagger. 2015. Swagger codegen: Generate clients, servers, and documentation from openapi specifications. https://swagger.io/tools/swagger-codegen/. Accessed: 2023-10-10.

Rails Core Team. 2005. Ruby on rails: A web-application framework for ruby. https://rubyonrails.org/. Accessed: 2023-10-10.

Yeoman Team. 2012. Yeoman: The web's scaffolding tool for modern webapps. https://yeoman.io/. Accessed: 2023-10-10.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):71.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand. Association for Computational Linguistics.

A Prompts

We provide the scoring prompts for the rating mechanism that were not included in the main text here in the appendix, demonstrating our evaluation criteria. The detailed prompts are shown in Tab. 7. Additionally, we provide the scoring prompt for the task completeness score evaluation criteria, as shown in Tab. 8.

B Corpus

We have provide detailed descriptions of the dataset (See Tab. 9). All six application codes have been thoroughly tested before we started our experiments.

Rating Prompt
Please rate the following code with a maximum score of 100 points based on the following criteria: 1. Functional completeness (whether all required functionalities in outline are implemented); 2. Whether there are obvious issues with imported libraries; 3. Whether parameter settings correspond to the Arduino code in outline (extract parameters such as pins or MQTT connection from the Arduino code, but do not directly copy its functions and libraries). 4. Whether the code compiles successfully. If the score is below 80 points, please specify the specific deficiencies. Additionally, don't take app.overlay and other configurations into account to evaluate the score. Points will NOT be deducted for missing MQTT communication or peripheral control code if the task specifies that these parts are NOT required. Don't deduct points because of warnings in the log! Don't deduct points because of warnings in the log! **Return the result in the following format:** –score–: <score between 0 and 100>, –feedback–: <specific deficiencies or improvements needed> Generated code: here is code Outline: here is outline Compilation Log: here is log Please provide the score and explain the deficiencies. You must obey the format including the notation "–" to extract the score and feedback. When generating the final output, do not include < and >. These angle brackets are only for illustrative purposes. If the compilation fails, you need to summarize the failure reasons and assist in improving the code.

Table 7: Prompt for Code Rating

Completeness Scoring Prompt
You should rate the following code, and you must obey the rating rule shown as below. You can only rate on a scale of 1, 2, 3, 4, or 5 based on the following criteria: 1. If there is only a main function or just text without even basic code, the score is 1; 2. If only part of the functionality is implemented, such as only communication or only peripheral control, the score is 2; 3. If the functionality is mostly implemented but lacks some specific operational code or uses comments in place of code(e.g., lack of the using of pwm library for lightness, lack of the using of adc library of smoke detect, lacking of the display for LED, lacking of the reading for dht), the score is 3; 4. If the functionality is mostly implemented without using comments as substitutes for code, but the implementation is not yet refined(e.g., lack of reconnection, lack of initialization, lack of closing and so on), the score is 4; 5. If all functionalities are fully and thoroughly implemented, the score is 5. **Return the result in the following format:** –score–: <score on a scale of 1, 2, 3, 4, or 5 >, –feedback–: <specific deficiencies or improvements needed> Task: here is task Code: here is code Must provide the score! You must obey the format including the notation "–" to extract the score and feedback. When generating the final output, do not include < and >. These angle brackets are only for illustrative purposes.

Table 8: Prompt for Code Rating

Table 9: Comparison of applications across Arduino, RIOT OS, and Zephyr OS.

Application	**Arduino (Lines)**	**Functional Description**	**RIOT OS (Lines)**	**Zephyr OS (Lines)**
Thermometer	132	Measures humidity/temperature via DHT11 sensor (ESP32-WROOM-32) and transmits data to cloud platform using MQTT/CoAP.	204	373
Doorbell	147	Implements a doorbell system using ESP32-WROOM-32, button, and buzzer. Triggers events when pressed and communicates status to cloud via MQTT/CoAP.	206	353
Switch	253	Dual-row switch controlling LED buttons. Supports bidirectional communication with cloud platform (MQTT/CoAP) for remote control.	228	412
SmartLight	213	Controls RGB LED (HW-478) with two buttons: Button1 cycles colors (red/green/yellow), Button2 adjusts intensity (on/strong/off). Cloud control via MQTT/CoAP.	244	418
SmokeDetect	162	Smoke detection system using ESP32-WROOM-32 and MQ2 sensor. Sends periodic air quality data and triggers alarms via MQTT/CoAP when thresholds are exceeded.	223	389
Display	272	Uses ESP32-WROOM-32 with I²C OLED to monitor temperature/humidity. Sends alerts for high temperatures and allows remote screen control via cloud (MQTT/CoAP).	262	366