AutoEmbed: LLM-driven Automated Software Development for Generic Embedded IoT Systems

Huanqi Yang
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
huanqi.yang@my.cityu.edu.hk

Mingzhe Li
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
mingzheli8-c@my.cityu.edu.hk

Mingda Han
Shandong University
Qingdao, China
mingdhan@mail.sdu.edu.cn

Zhenjiang Li
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
zhenjiang.li@cityu.edu.hk

Weitao Xu*
Department of Computer Science
City University of Hong Kong
Kowloon Tong, Hong Kong
weitaoxu@cityu.edu.hk

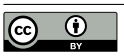
Abstract

Embedded system development is crucial for enabling seamless connectivity and functionality across a wide range of Internet of Things (IoT) applications. However, such a complex process requires cross-domain knowledge of hardware and software and hence often necessitates direct developer involvement, making it labor-intensive, time-consuming, and error-prone. To address this challenge, this paper introduces `AutoEmbed`, the first automated software development platform for general-purpose embedded IoT systems. The key idea is to leverage the reasoning ability of Large Language Models (LLMs) and embedded system expertise to automate the hardware-in-the-loop development process. The main methods include a component-aware library resolution method for addressing hardware dependencies, a library knowledge generation method that injects utility domain knowledge into LLMs, and an auto-programming method that ensures successful deployment. We evaluate `AutoEmbed`'s performance across 71 modules and four mainstream embedded development platforms with over 350 IoT tasks. Experimental results show that `AutoEmbed` can generate codes with an accuracy of 95.7% and complete tasks with a success rate of 86.5%, surpassing human-in-the-loop baselines by 15.6%–37.7% and 25.5%–53.4%, respectively. We also show `AutoEmbed` 's potential through case studies in environmental monitoring and remote control systems development.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; • **Hardware** → *Communication hardware, interfaces and storage*; • **Software and its engineering** → **Software development techniques**.

*Corresponding author



SenSys '26, Saint Malo, France

ACM ISBN 979-8-4007-2309-4/26/05
https://doi.org/10.1145/3774906.3800466

Keywords

Embedded Systems, Internet of Things, Large Language Models, Automated Software Development, Code Generation

ACM Reference Format:
Huanqi Yang, Mingzhe Li, Mingda Han, Zhenjiang Li, and Weitao Xu. 2026. AutoEmbed: LLM-driven Automated Software Development for Generic Embedded IoT Systems. In *ACM/IEEE International Conference on Embedded Artificial Intelligence and Sensing Systems (SenSys '26), May 11–14, 2026, Saint Malo, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3774906.3800466

1 Introduction

1.1 Background and Motivation

Embedded systems support numerous Internet of Things (IoT) applications, with the market projected to reach $258.6 billion by 2032 [55]. These systems play key roles across various industries, including healthcare [25, 53, 78], agriculture [21, 69], smart cities [59, 62], human sensing [33, 81, 82], and secure communications [80, 83, 84]. For example, in a smart city [46, 57], embedded systems control street lighting and traffic signals based on sensor data to optimize energy use and traffic flow.

Developing such systems straddles the hardware-software interface, demanding a cross-domain understanding of how devices interact with the physical world [37]. Specifically, this intricate process involves several labor-intensive steps. Despite its potential, the process of developing embedded systems is inherently constrained by several labor-intensive steps. Initially, developers must manually address dependencies by installing and configuring the essential libraries for hardware modules. Next, they write code to define the device's behavior, considering interactions with various sensors, actuators, and communication modules. Afterward, the code is compiled, which often reveals configuration or dependency errors requiring further refinement. Once the code is successfully compiled, it needs to be uploaded to the development platform for testing and deployment. Currently, the development of embedded systems predominantly relies on manual processes using traditional Integrated Development Environments (IDEs) (*e.g.*, Eclipse, Keil µVision, and IAR). However, the variety and complexity of programming tools lead to high learning and development costs for

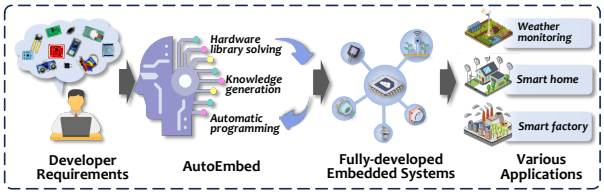


Figure 1: Motivation for `AutoEmbed`.

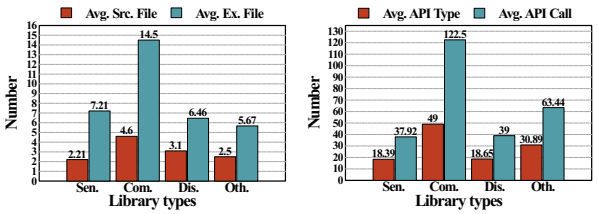
developers [8, 58]. Additionally, the diversity of hardware modules (*e.g.*, sensors, displays, and communication modules) further increases the complexity, making manual development even more challenging. In summary, the traditional embedded system development process is labor-intensive, time-consuming, and error-prone.

This paper introduces a novel automation approach that leverages Large Language Models (LLMs) to streamline embedded IoT system development. Building on the recent success of LLMs (*e.g.*, GPT-4 and Claude-3.5), we aim to harness their capabilities to simplify the development process by automating processes such as dependency-solving, coding, and deployment. Although there are some existing coding automation tools, such as those utilizing LLMs for general coding tasks [13, 48, 89], they only provide partial relief by assisting in tasks like code generation and debugging. For instance, CODET [13] aids in generating code with built-in tests, while LDB [89] verifies runtime execution step by step to handle complex logic flow and data operations. However, they fall short in driving specific hardware devices (*e.g.*, microcontrollers and sensors) due to a lack of hardware-specific knowledge in embedded systems, such as peripheral interface configurations and library dependencies. Consequently, these tools cannot fully address the intricacies and specialized requirements of embedded system development. Moreover, recent work [26, 30] investigates the role of LLMs in supporting embedded system programming, while Xu et al. [77] explore their application in embedded system programming and circuit design through simulations. However, these studies are limited to human-in-the-loop user studies or simulations, covering a restricted set of tasks and modules, and fall short of achieving full automation. To address this gap, we propose an LLM-powered automation system that streamlines the dependency-solving, programming, and deployment processes in embedded system development, resulting in developed embedded systems for various IoT applications as illustrated in Fig. 1.

While individual embedded system components often come with sample code and documentation, the true challenge lies in their **composition and integration**. The value of LLM-based automation extends beyond simple code generation to include: (1) automatically composing building blocks from different hardware vendors with varying API designs, (2) resolving subtle incompatibilities between libraries and development platforms, and (3) managing the complex dependency resolution process across thousands of available libraries. This automated composition and integration capability significantly reduces developer effort even when individual component examples exist, as it eliminates the need for manual trial-and-error integration and debugging processes.

1.2 Challenges and Contributions

We need to tackle several key challenges to achieve the goal.



(a) File count. (b) API count.

Figure 2: Library complexity distribution. (a) Source and example files, and (b) API types defined in source and API calls in the example. (Sen., Com., Dis., Oth.: Sensors, Communication modules, Displays, Others.)

Challenge 1: Diversity in Hardware Dependency. Various hardware components, such as communication modules, sensors, and displays, are built on different architectures and perform specialized tasks. Each component relies on specific library dependencies to function effectively, leading to unique challenges in dependency resolution. As noted on the Arduino library website [5], there are over 7,000 libraries across diverse categories, including 1,527 communication libraries, 1,285 device control libraries, 1,435 sensor libraries, and so on. Therefore, accurately identifying and selecting the essential libraries for different hardware components is a significant challenge. To address this issue, we explore the fundamental principles of library selection, revealing that different libraries exhibit distinct compatibility with specific models and varying support for development board architectures. Additionally, we find that a high version iteration frequency suggests active maintenance and ongoing improvements, reflecting the library's stability and functionality. Based on these findings, we propose an automated dependency solving method that can efficiently identify the most suitable libraries for specific hardware components.

Challenge 2: Lack of Library Knowledge. Embedded systems development requires specialized library knowledge, which standard LLMs are not inherently equipped to handle. Moreover, the diversity of libraries across different modules (*e.g.*, Adafruit SSD1306 library for OLED displays or the FastLED library for LED strips [4, 6]) and the variation in their usage further complicate the issue. As shown in Fig. 2, different libraries include numerous source and example files that enhance developer understanding by providing extensive API definitions and usage examples. This underscores the need for a robust method to enhance LLMs with library knowledge. Specifically, different libraries often have distinct APIs, each with varying parameters, return values, and usage patterns. Correctly interfacing with these libraries requires a deep understanding of their specific designs. Standard LLMs may generate code that is syntactically correct but contextually inappropriate due to a lack of specialized library knowledge. To address this challenge, we propose a knowledge generation method that extracts and injects library API and utility knowledge into the LLM's memory, enabling syntactically and contextually appropriate solutions.

Challenge 3: Complexity of Embedded System Programming. Programming is a crucial part of embedded systems development, involving coding, compiling, and flashing, with each step requiring precise execution [9, 28]. Fig. 3 illustrates the differences between general-purpose programming and embedded system programming. In general-purpose programming, the workflow typically involves coding, debugging, and deployment. In contrast,

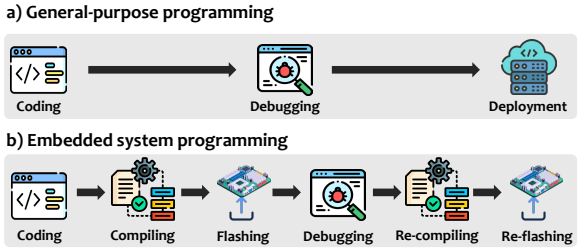


Figure 3: Programming pipeline comparison.

embedded system programming introduces additional steps such as compiling and flashing, which require specialized configurations and are particularly error-prone. For example, using an incorrect microcontroller configuration or selecting an incompatible library can result in compilation failures, while coding errors may lead to functional verification failures after flashing. Therefore, the inherent complexity of these steps demands an advanced programming mechanism. To tackle this challenge, we propose an automated programming process that incorporates two nested reasoning and acting feedback loops: a compile loop and a flash loop. Specifically, the compile loop ensures that the correct configurations and libraries are applied before each flashing cycle, while the flash loop verifies that the embedded system functions accurately and completely as described in the user tasks.

By incorporating the above solutions, we design and implement `AutoEmbed` (**Demo:** https://autoembed.github.io/#Demo) , a comprehensive framework that fully automates the dependency-solving, programming, and deployment processes for embedded system development. This advancement lowers the barrier to entry for embedded system development, pushing it into practical, real-world applications. Our extensive evaluation, involving over 70 hardware modules, four development platforms, and over 350 IoT tasks, demonstrates that `AutoEmbed` achieves an average coding accuracy of 95.7% and an average completion rate of 86.5%, highlighting the effectiveness of `AutoEmbed`. We present real-life case studies to demonstrate how `AutoEmbed` benefits the development of complex systems: 1) an environmental monitoring system and 2) a remote control system. The results show that `AutoEmbed` can develop these systems in 2.6 min and 3.1 min, respectively.

Our contributions are summarized as follows:

- To the best of our knowledge, `AutoEmbed` is the first automated embedded system development tool that significantly reduces development time and minimizes errors.
- We introduce a novel suite of methods, including a library resolution method for hardware dependency solving, a knowledge generation method that enhances LLMs with specialized knowledge, and an auto-programming method for successful deployment.
- Experimental results show that our system achieves an average coding accuracy of 95.7% across various IoT tasks. We also conduct real-life case studies to show `AutoEmbed` 's potential.

2 Preliminaries

2.1 Embedded System Development

Assume an embedded system $E_s = g(D, M)$, where D represents the development platform, and M refers to the connected modules (*e.g.*, sensors, communication modules, and displays). The system E_s is the result of combining D and M. The pipeline of embedded system development is shown in Fig. 4.

Compiler Setting. The first step in the development process is configuring the settings within the development environment to align with the specific characteristics of the target system E_s. This configuration process can be formulated as $C_h = f(E_s, P)$, where C_h represents the configured hardware settings, and P refers to the hardware-level configuration parameters (*e.g.*, pin mappings).

Solving Dependencies. Following the setting, solving dependencies of E_s becomes crucial to ensure smooth integration and functionality. This involves searching for and incorporating the necessary libraries that support the interaction between the development platform and the attached modules. This process can be expressed as $L_s = S(L_a, E_s)$, where L_s represents the selected libraries that satisfy the dependencies, L_a is the set of available libraries, and the search S is conducted based on the characteristics of E_s.

Coding. Afterward, the developers write the code to implement the desired functionality of the embedded system. Guided by L_s and C_h, the coding process is $G = h(T, C_h, L_s)$, where G represents the code, T is the task.

Compiling and Flashing. The next step is compiling the code into executable machine code that the hardware can understand. The compilation process is formulated as $B = c(G)$, where B is the resulting binary file. Once the code is compiled, the next step is flashing the binary onto the hardware. This step involves transferring the executable machine code onto the embedded system via a suitable interface (*e.g.*, USB, JTAG, or over-the-air [65, 66]). The flashing process can be expressed as $F = f(B, E_s)$, where F represents the flashed system.

Finally, the entire embedded system development process can be expressed as $F = \Phi(E_s, P)$. These processes are typically manually, making them labor-intensive, time-consuming, and error-prone.

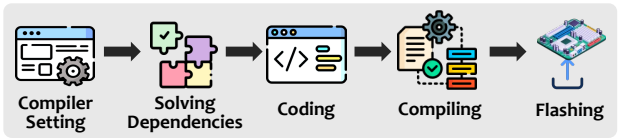


Figure 4: Embedded system development pipeline.

2.2 Large Language Models

LLMs typically refer to Transformer-based models [70] with billions of parameters, trained on extensive text datasets. Examples include models such as ChatGPT [49], GPT-4 [1], PaLM [17], and LLaMA [68], among others. These models demonstrate advanced capabilities not found in smaller models, including mathematical reasoning [19], program synthesis [14], and complex multi-step reasoning [72]. An LLM's input is a prompt, which is tokenized into a sequence of tokens, consisting of words or subwords, before processing. The inference process in LLMs can be represented as

$$P(y|x;\theta) = \prod_{t=1}^{T} P(y_t|y_{<t}, x;\theta), \tag{1}$$

where x is the input prompt, $y = (y_1, y_2, \ldots, y_T)$ is the sequence of tokens generated by the model, and θ represents the model parameters. The term $P(y_t|y_{<t}, x;\theta)$ denotes the probability of generating the token y_t given the previous tokens $y_{<t}$ and the

input x. This probability is computed using a softmax function over the model's output logits:

$$P\left(y_t \mid y_{<t}, x; \theta\right) = \frac{\exp\left(e_{y_t} \cdot h_t\right)}{\sum_{y' \in V} \exp\left(e_{y'} \cdot h_t\right)}, \tag{2}$$

where e_{y_t} is the embedding vector of the token y_t, h_t is the hidden state at time step t, which is a function of the previous tokens $y_{<t}$ and the input x, and V is the vocabulary set.

3 System Design

As shown in Fig. 5, AutoEmbed comprises two phases, preparation and execution phase. During the preparation phase, AutoEmbed initializes hardware configurations and solves library dependencies, then generates knowledge based on the library content. In the execution phase, AutoEmbed queries memory-augmented LLMs to generate task prompts and automate system programming.

3.1 Initialization

3.1.1 Hardware Configuration. AutoEmbed collects and standardizes essential metadata related to hardware information, ensuring seamless integration and minimal setup costs. Users simply input the type of hardware components, including the development platforms D and modules M, along with their specific pin assignments P. For example, as shown in Fig. 6, a user might specify the development platform as an Arduino Uno, with an LED connected to Pin 13, a button to Pin 7, and a DHT11 sensor to Pin 5. These inputs are the minimal necessary information required from the user, as they depend on the specific choices made for their project. This streamlined interaction minimizes user effort, allowing the system to identify and integrate the components.

3.1.2 Solving Library Dependencies. With the provided hardware information, traditional methods rely on users to leverage their experience or manually search for the appropriate library dependencies for each hardware component. However, this approach is inefficient and complex. To address this, we propose an automated library dependency solving method that can efficiently identify the most suitable libraries. The algorithm is shown in Alg. 1. AutoEmbed begins by extracting component information from the provided hardware metadata. It then performs a search to identify potential libraries, retrieving the top N libraries for further evaluation. The evaluation process involves assessing each library based on three criteria: **1) Name Match:** This is calculated using cosine similarity between the component name and the library's name, including its description and any related paragraphs. **2) Version Count:** The score is based on the number of available library versions, normalized between 0 and 1. **3) Architecture Compatibility:** This score is determined by whether the library supports the target hardware architecture. It receives a full score if the library is compatible and zero if it is not. Libraries are ranked based on these scores, and the library with the highest overall score is selected for use. If a library's architecture compatibility score is zero, it is deemed unsuitable and excluded from consideration.

Our heuristic-based scoring method provides a more scalable and robust solution by automatically evaluating libraries based on their intrinsic characteristics, enabling the system to adapt to new hardware and library ecosystems without manual intervention. Additionally, we consider version counts as a key factor in our selection process. Libraries with recent updates and active maintenance are prioritized, as this approach significantly mitigates the impact of outdated practices and poorly maintained libraries on the system's performance.

Algorithm 1: Library Selection Algorithm.

Initialize: *comps*, *tgt_arch* // Components list, platform architecture
final_libs ← {}; // Initialize final libraries
for $c \in$ *comps* **do**
  search_results ← CLISearch(c); // Perform CLI search
  top_libs[c] ← TopNLibs(search_results, N); // Select top N libraries
end
for $c \in$ *comps* **do**
  best_lib ← ∅; // Initialize best library
  best_score ← $-\infty$; // Initialize best score
  for $l \in$ *top_libs[c]* **do**
    details ← ParseLibDetails(l); // Parse library details
    $m \leftarrow \text{CosSim}(details["Name"], c)$; // Name match score
    $v \leftarrow \frac{\text{Count}(details["Vers"])}{\text{MaxVerCount}(top\_libs[c])}$; // Version count score
    $a \leftarrow \text{ArchScore}(details["Arch"], tgt\_arch)$; // Architecture score
    $score \leftarrow (m + 0.1 \cdot v + 0.1 \cdot a) \cdot a$; // Compute total score
    if $score > best\_score$ **then**
      best_score ← $score$; // Update best score
      best_lib ← l; // Update best library
    end
  end
  final_libs[c] ← best_lib; // Assign best library
end
return final_libs; // Return final libraries

3.2 Knowledge Generation

3.2.1 Library Knowledge Extraction. Once the most suitable libraries are identified, AutoEmbed needs to learn how to use them. This involves two main steps: API extraction and obtaining API usage information. First, AutoEmbed identifies the selected library for each component, searches for header (.h) files within the library, and uses the LLM to extract API declarations, summarizing them into the API table. Then, it searches for example (.ino) files within the same library and extracts knowledge on how to use the extracted APIs (e.g., orders, parameters, and return values). This information is further integrated into the API table. As shown in the purple section in Fig. 7, the library knowledge generator extracts the API table content from the header and example files using LLM. This process extracts knowledge about how to effectively utilize the libraries. Tab. 1 shows an example entry of the API table of the DS18B20 component.

3.2.2 Functionality Knowledge Understanding. To effectively utilize each component, AutoEmbed creates a component utility table in memory. This table summarizes the functionality and API sequences associated with each component. As shown in the yellow

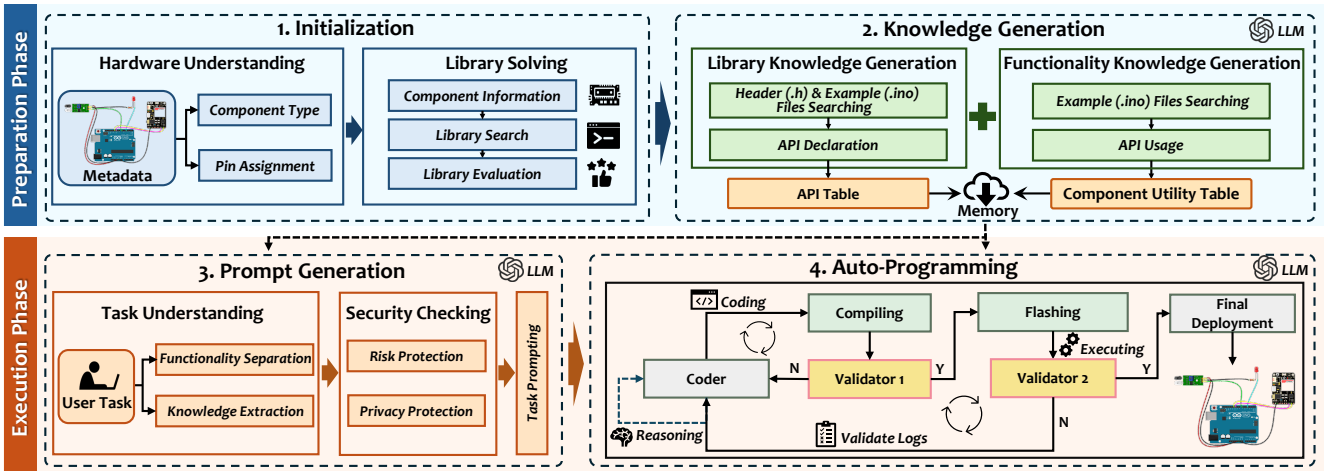


Figure 5: AutoEmbed Overview.

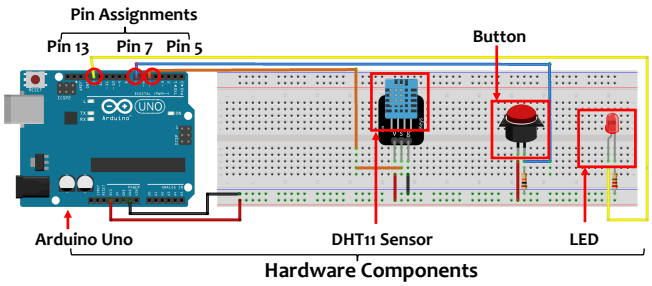


Figure 6: Hardware configuration.

API Table Example (1 item)

```
DS18B20.h:
- name: getTempC
description: Get temperature in Celsius
parameters: []
return_type: float
practices:
API order: "#include <DS18B20.h>, DS18B20 ds(2);"
parameter usage: "float tempC = ds.getTempC();"
return value handling: "if (tempC != -127.0) { ... }"
```

Table 1: API Table Example. Detailed API table examples are available at https://autoembed.github.io/#Example.

section in Fig. 7, the functionality knowledge generator extracts functionalities from example files by querying the LLM to summarize the API sequences.

Specifically, we summarize the functionality of the examples and identify the API calls related to the component. This forms a utility table, where entries represent the functionalities F and their associated APIs A. After analyzing all example codes, we obtain a utility table in memory containing n entries, where n represents the total number of examples. Each entry in the table corresponds to an example and is divided into two parts: ⟨Functionality, API⟩. **Functionality:** Represents the functionality F_i. It can be perceived as a task that can be completed using these APIs. **API:** Represents the sequence of APIs A_i^j used from the initial API to the last. This table provides information about the required operations to achieve each functionality, aiding AutoEmbed in planning how to complete a task. Tab. 2 shows an example entry of the table.

Our API table-based knowledge injection method differs from standard Retrieval-Augmented Generation (RAG) approaches in several key aspects that are critical for embedded system development: (1) **Structured semantic understanding:** Our approach captures the specific semantics, parameter types, return values, and usage patterns of hardware APIs in a structured format, enabling the LLM to better understand the precise requirements for generating functionally correct embedded code. (2) **Hardware-specific context:** Unlike general-purpose RAG systems that retrieve text passages, our method provides structured information about API ordering, parameter constraints, and hardware-specific usage patterns that are essential for embedded programming. (3) **Compilation and deployment awareness:** The structured API tables include information about compilation dependencies and deployment considerations that are crucial for successful embedded system development but may not be captured effectively in unstructured text retrieval. This targeted approach ensures that the LLM receives the precise, structured information needed for embedded system code generation rather than general textual descriptions. While we use static parsing for knowledge extraction, the auto-programming loop provides an implicit form of runtime verification. If the generated code fails during compilation or execution, the system iterates, which helps to filter out incorrect API usage patterns.

Component Utility Table Example (1 item)

```
DS18B20.h:
- name: Alarms.ino
functionality: handles temperature alarms,
sets alarm thresholds,
and prints temperature data
API: [ "selectNext", "setAlarms", "doConversion",
"selectNextAlarm", "getAlarmLow",
"getAlarmHigh", "getTempC" ]
```

Table 2: Component utility table example. Detailed examples are available at https://autoembed.github.io/#Example.

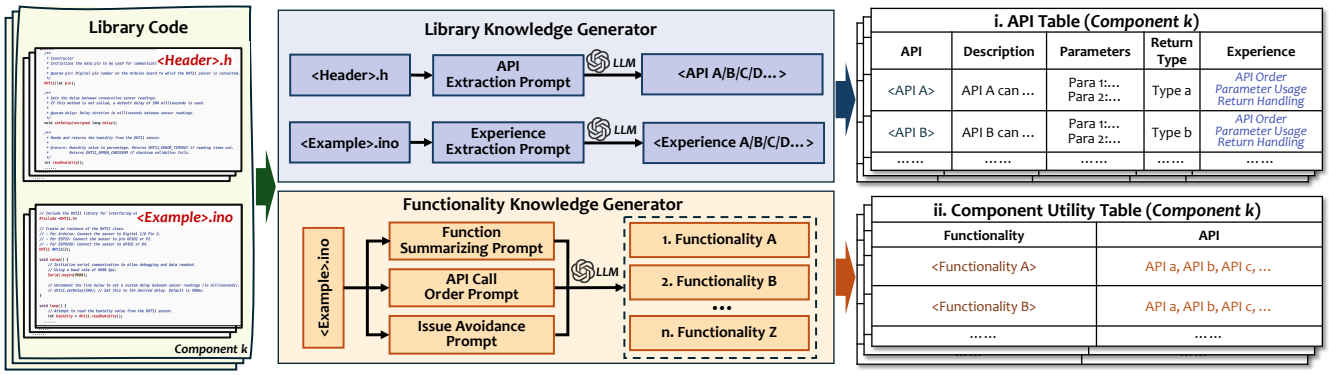


Figure 7: Knowledge Generation.

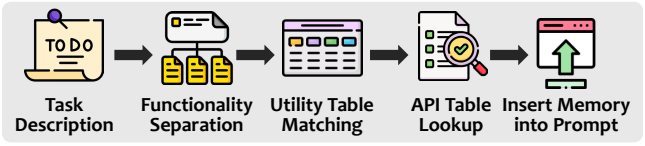


Figure 8: Selective memory pick-up process.

3.3 Prompt Generation

3.3.1 Task Understanding and Memory Pick-up. To leverage memorized knowledge from chosen libraries, directly embedding pertinent header or example file content into the prompt to guide the LLM may surpass the token limit, such as the 4096 tokens for GPT-4. Moreover, this approach can escalate costs. Hence, we opt to incorporate only the most crucial information necessary for accomplishing the user's task. To address this, we devise a selective memory pick-up method. The workflow is shown in Fig. 8, with details below. **Functionality Separation.** We query the LLM to separate the task description into functionalities corresponding to each component. For instance, given the task "Record the DHT11 temperature reading to SD card," the identified functionalities would be: "initialize DHT11 sensor," "initialize SD card," "read temperature from DHT11 sensor," and "store data to SD card." **Knowledge Extraction.** 1) Utility Table Matching: For each functionality in the task, we match it to functionalities in the memory's utility table by comparing their similarity. We use the Term Frequency-Inverse Document Frequency (TF-IDF) method [2] to represent natural language sentences as weighted vectors. The cosine similarity between the vectors of the functionality in utility table F_t and the current functionality C_t is denoted as $\text{sim}(E(F_t), E(C_t))$. We then identify the k most similar functionalities in the memory, denoted as $\{F_{t1}, F_{t2}, \ldots, F_{tk}\}$. For each F_{ti}, we retrieve the corresponding APIs from the utility table. 2) API Table Lookup: Next, we look up the API table to retrieve usage information about the matched APIs. This information A_T is then incorporated into the prompt, allowing for more accurate and relevant code generation.

3.3.2 Security Checking. **1) Risk Protection.** During embedded system development, certain actions may lead to system malfunctions or disrupt normal device operation. These actions are considered risky and require developer confirmation. For example, a developer may adjust the PWM signal frequency to control a motor's speed. If the frequency is set incorrectly, it could cause the motor to overheat or even get damaged. To mitigate such risks, the system prompts the developer for confirmation before executing these high-risk operations. The system will ask: *"This action could lead to overheating or potential damage to the motor, please answer requires_confirmation=Yes."* Additionally, key phrases like "warning" are jumped to flag potentially risky actions. The system proceeds with the action only after receiving confirmation from the developer. **2) Privacy Protection.** It is crucial to protect sensitive information such as device IDs, network credentials, and user names. We add a privacy filter that can mask private information in user queries. For example, if a user query includes *"Set device ID to 12345 and Wi-Fi password to 'password123',"* the system detects this Personal Identifiable Information (PII). The PII is then replaced with non-private placeholders before sending the query to the cloud, resulting in *"Set device ID to <device_id> and Wi-Fi password to <password>."* After receiving the response from the cloud, the system maps the placeholders back to the original sensitive information and executes the action.

3.3.3 Task Prompting. To effectively guide the LLM, we construct a structured prompt that includes the user-defined task, configuration metadata for the involved hardware components, relevant API usage information, and coding rules. For a concise representation of the prompt design, refer to Table 3. This structured prompt ensures the LLM generates accurate and optimized code while staying within token limits and reducing costs.

AutoEmbed Task Prompt (Coder)

You are an expert in embedded systems programming.
Task Description
{'type': 'task', 'id': 'task'}
Components Matedata
{'type': 'board', 'id': 'board_name'}
{'type': 'component', 'id': 'component_name'}
{'type': 'library', 'id': 'Library_name'}
{'type': 'pin connections', 'id': 'pin_connections}
API Usage Information (Memory)
Header file and APIs available and relevant for this task
Rules
- Include debug statements.
- Use only provided APIs.
- Follow embedded C/C++ standards.

Table 3: Prompt design. Detailed prompt templates and example available at https://autoembed.github.io/#Prompt.

Algorithm 2: Auto-Programming Algorithm.

Initialize: $T, M_c, M_e, M_{v1}, M_{v2}$; // Task, Coder, Executor, two Validators
Set $FL_c, FL_f \leftarrow [\,]$; // Initialize feedback Log
Set $t_f = 0$; // Initialize flash debug trial count
repeat
  Set $t_c = 0$; // Initialize compile debug trial count
  repeat
    Generate Code:
    $G \leftarrow M_c(T, FL_c, FL_f)$; // Generate code with DEBUG INFO inserted
    Compile Code:
    $O_c, B \leftarrow M_e(G)$; // Compile code, get output and binary file
    $t_c \leftarrow t_c + 1$; // Increment trial count
    Validate Compile:
    $FL_c \leftarrow M_{v1}(O_c)$;
  until $FL_c == Success$ *or* $t_c \geq$ *max flash debug trials*;
  if $FL_c == Success$ **then**
    Flash Code:
    $O_f \leftarrow M_e(B)$; // Flash the binary file and get the debug output
    Validate Execution:
    $FL_f \leftarrow M_{v2}(O_f, T)$
    $t_f \leftarrow t_f + 1$; // Increment trial count
  end
until $FL_f == Success$ *or* $t_f \geq$ *max compile debug trials*;
$G_f \leftarrow G - D(G)$; // Clean final code
return G_f

3.4 Auto-Programming

The auto-programming method plays a crucial role in our system, guaranteeing that the produced code is syntactically correct and functionally accurate. This process involves multiple validation strategies: (1) compilation error detection and correction, (2) runtime subtask behavior verification through debug output analysis, and (3) task logic verification by comparing debug log output with expected execution flow. It functions within two nested dynamic reasoning and action loops, as shown in Fig. 5. The algorithm design is detailed in Alg. 2.

3.4.1 Coder. The Coder begins by receiving a task prompt (see Tab. 3) from the Prompt Generation process to generate code. Guided by the prompt described in Sec. 3.3.3, the coding rules include integrating DEBUG INFO into the code to monitor various aspects of the execution state, such as order and logic. To facilitate automated monitoring, DEBUG INFO is dynamically generated by the LLM in conjunction with code generation. Let $D(G)$ represent the set of DEBUG INFO annotations. For each subtask T_i in G, DEBUG INFO statements D_i are included such that $D(G) = \bigcup_{i=1}^{n} D_i$, where n is the total number of subtasks. Our auto-programming method advances beyond basic compilation trials by adopting a systematic approach to code generation and validation. This method integrates three key stages: (1) *Compilation Error Handling*, where the system iteratively identifies and resolves syntax errors, missing includes, and library compatibility issues using large language model (LLM) queries; (2) *Runtime Behavior Verification*, which examines debug output to confirm that each subtask executes correctly; and (3) *Task Logic Verification*, which compares the debug log's output sequence with the expected output to validate the subtask execution flow and logic. These stages are orchestrated through two nested loops: the compile loop and the flash loop.

3.4.2 Compile Loop. The compile loop begins by passing the generated code to the compiler, which checks for syntax errors, optimizes the code, and translates it into machine-readable instructions. Afterward, the compilation result is sent to query the Compile Validator, which queries the LLM to check if the compilation was successful. If unsuccessful, it returns summarized logs to the Coder for error correction. If successful, the code is deemed ready for flashing.

3.4.3 Flash Loop. In the flash loop, the compiled code is flashed to the development platform for execution. The Flash Validator leverages the LLM to utilize the embedded DEBUG INFO to identify and correct errors by performing two checks: the **Order Check**, which ensures that for each subtask T_i, the start time precedes the end time ($\forall T_i, ; t_{\text{start}}(T_i) < t_{\text{end}}(T_i)$) and that the execution order follows the expected sequence; and the **Logic Check**, which verifies that the internal logic of each subtask T_i adheres to the expected behavior and constraints, ensuring correct functionality beyond just task order and timing. If discrepancies are detected, the Flash Validator provides feedback to the Coder for code refinement and restarts the loop until the code passes validation. While we use static parsing for knowledge extraction, the auto-programming loop provides an implicit form of runtime verification. If the generated code fails during compilation or execution, the system iterates, which helps to filter out incorrect API usage patterns.

Fig. 9 presents an example involving the reading of temperature and humidity data from a DHT11 sensor and the subsequent control of an LED based on the temperature readings. The process is detailed as follows. The Coder generates the initial code G with embedded DEBUG INFO statements $D(G)$ (step ❶). The code is then compiled, and its readiness for deployment is evaluated. If the code is not ready, the Compile Validator identifies the issues and prompts corrections. In this scenario, the code passes the validation successfully (step ❷). Then it flashs the code to the embedded development board and collects the execution logs. Subsequently, the Flash Validator analyzes these logs to assess the correctness of the code. In this scenario, the Flash Validator identifies a logical error where the LED is incorrectly turned off when the temperature exceeds 30°C. It provides feedback to the Coder for necessary code correction (step ❸). Upon receiving the feedback, the Coder evaluates and implements the required changes to correct the LED control logic in the code G. Additionally, the Coder updates the corresponding DEBUG INFO statements $D(G)$ to reflect the modifications. (step ❹) The process is repeated iteratively until the code is verified to be successfully compiled and error-free (step ❺–❻). Finally, the DEBUG INFO are removed, and the clean code is redeployed to the development board (step ❼–❾). Auto-Programming integrates the LLM programmer, Executor, and Validator to create a robust feedback loop, ensuring the generated code is correct.

4 Evaluation

4.1 Experimental Setup

`AutoEmbed` Implementation. The automation framework leverages HTTP requests to interface with LLM APIs, enabling seamless interaction with models such as GPT-4, with GPT-4o set as the default model. It was developed using Python 3.9 and is fully integrated with the Arduino CLI compiler, a tool that offers a universal

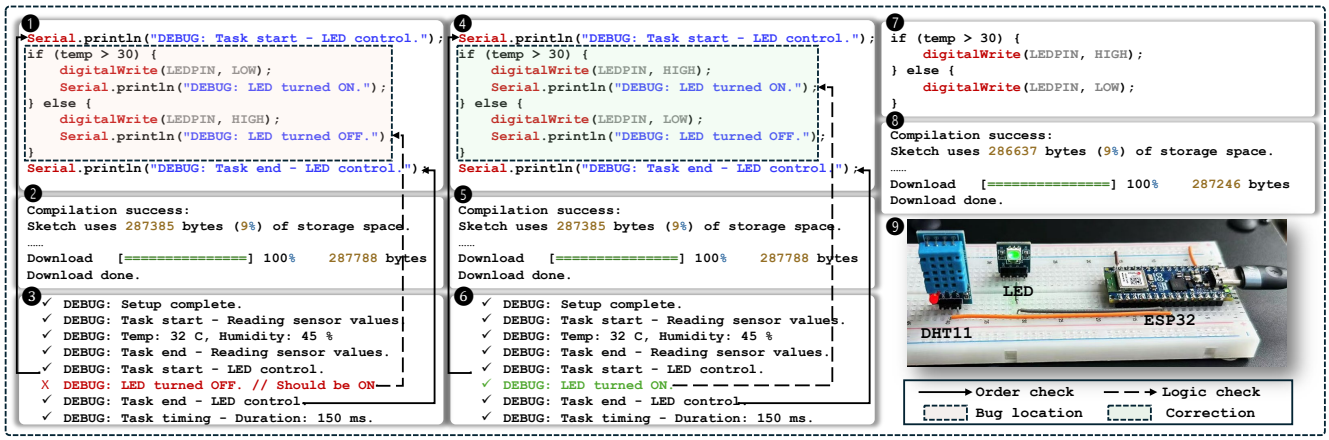


Figure 9: Auto-programming example.

Figure 10: Experimental devices. a) Development platforms and b) hardware modules. Detailed list available at https://autoembed.github.io/#Hardware.

Table 4: Development platform specifications.

Platform	MCU*	Max. freq.	FLASH	SRAM	Size
Uno R3	ATmega328P	20MHz	32KB	2KB	68.6×53.4mm
NUCLEO-L4	STM32L432KC	80MHz	256KB	64KB	50.3×18.5mm
Nano RP2040	RP2040	133MHz	16MB	264KB	45.0×18.0mm
Nano ESP32	ESP32-S3	240MHz	32MB	512KB	43.1×17.8mm

*AT: Atmel, STM: STMicroelectronics, RP: Raspberry Pi Pico, ESP: Espressif.

way to interact with the development environment from the command line. Prompt design: https://autoembed.github.io/#Prompt.

Development Platforms. To comprehensively evaluate the performance of AutoEmbed, we selected four popular development platforms that are highly representative in consumer electronics and education: Uno R3, NUCLEO-L4, Nano RP2040, and Nano ESP32, as shown in Fig. 10 (a). The specifications of these platforms are shown in Tab. 4. These boards showcase diverse MCU architectures, varying computational capabilities, and distinct application scenarios, effectively representing the majority of embedded platforms.

Modules. To demonstrate the extensive compatibility of our system, 71 modules were utilized to validate its functionality across a wide range of embedded devices and modules. As shown in Fig. 10 (b), we display most of the modules used. Some similar hardware was omitted due to space constraints. These modules encompass environmental sensors (*e.g.*, temperature, humidity, atmospheric pressure), motion sensors (*e.g.*, magnetic angle sensors, accelerometers), communication modules (*e.g.*, Bluetooth, LoRa), display modules, control devices, and others, thereby representing a comprehensive array of devices. For each module, various tasks of differing complexity levels were used to assess their integration.

Task Dataset. We introduce EmbedTask, an IoT task dataset designed to evaluate the effectiveness of a automated embedded system development process from start to finish. EmbedTask includes 355 tasks, covering different modules and varying complexity levels. These tasks span a range of applications, such as environmental monitoring, motion detection, data communication, digital display, and motion control. EmbedTask classifies tasks into three levels: Level 1 tasks assess basic functionality of a single module (*e.g.*, reading temperature); Level 2 tasks involve additional modules or logic integration (*e.g.*, triggering an LED based on humidity threshold); Level 3 tasks require collaboration between multiple modules with advanced logic (*e.g.*, logging sensor data to SD card while controlling a servo motor). Task complexity is formally defined as the product of the number of functionalities (N_f) and the number of components (N_c) involved: Task Complexity $= N_f \times N_c$. This metric aligns with established system engineering principles, where complexity increases with both functionalities and components [10]. As illustrated in Fig.11, the variation in task complexity arises from differences in the number of functionalities and components across difficulty levels. Tab. 5 shows examples of tasks in three difficulty levels for LTR390 sensor.

Compared with existing benchmark [26], our experimental setup involves 30× more tasks, 5× more modules, and 2× more development platforms. This coverage ensures that our system is capable of addressing a wide range of IoT tasks with diverse hardware.

Metrics. To evaluate AutoEmbed's performance, we consider a sequence of functionalities $\{F_1, F_2, \ldots, F_m\}$ alongside a sequence of API usages $\{A_1, A_2, \ldots, A_n\}$ performed by human annotators to complete a task T. If AutoEmbed can use a sequence of APIs $\hat{A} = \{\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_n\}$ corresponding to each functionality, we employ the following two metrics:

(1) **Coding Accuracy:** This metric is defined as the ratio of API usage $\hat{A}_i$ that matches the reference A_i defined in library, expressed as $P(\hat{A}_i = A_i)$. An API usage is deemed correct if both the API and its parameters are accurate.

Task example (LTR390)

Component: LTR390 UV Sensor

Difficulty Level 1 (Task Complexity = $N_f \times N_c = 2 \times 1 = 2$)

Task brief: Read UV index and print to serial monitor.
Task Detail: I want to read the UV index using the Adafruit LTR390 and print it to the serial monitor.

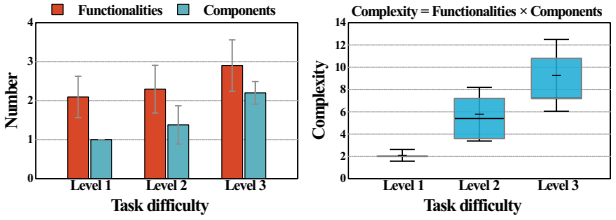
Difficulty Level 2 (Task Complexity = $N_f \times N_c = 3 \times 2 = 6$)

Task brief: Log UV data only if UV index exceeds 3, every 5 seconds.
Task Detail: I want to log UV data to an SD card using the Adafruit LTR390 and an SD module, but only if the UV index is greater than 3, and I want to do this every 5 seconds.

Difficulty Level 3 (Task Complexity = $N_f \times N_c = 3 \times 3 = 9$)

Task brief: Implement a UV alarm system, if UV index > 5, turn on a warning LED and buzzer.
Task Detail: I want to set up a UV alarm system using the Adafruit LTR390 that monitors the UV index continuously. If the UV index exceeds 5, it should activate an alarm consisting of an LED and a buzzer.

Table 5: Task examples for LTR390 UV sensor



(a) Number. (b) Complexity.

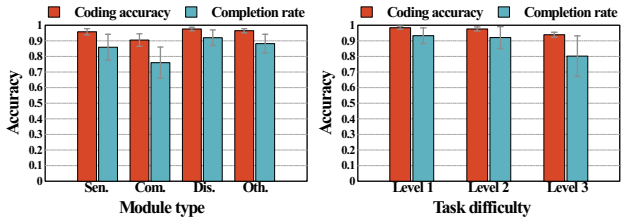
Figure 11: Complexity of `EmbedTask`.

(2) **Completion Rate:** This is the probability that the system completes all functionalities in task correctly within one attempt, represented as $P(\hat{A} = A)$. This metric indicates the likelihood of successfully completing a task.

4.2 Overall Performance

Overall Accuracy. Fig. 12 presents the accuracy percentages for task execution across various modules and task difficulty levels using `AutoEmbed`. Specifically, coding accuracy is 95.8%, 90.5%, 97.6%, and 96.5%, with corresponding completion rates of 85.9%, 76.1%, 92.2%, and 88.2% for the sensor, communication, display, and other module types, respectively. The communication modules demonstrate lower accuracy due to the increased complexity of their associated libraries, as shown in Fig. 2. Across difficulty levels, coding accuracy stands at 98.4%, 97.6%, and 93.9%, with completion rates of 93.3%, 92.1%, and 80.2% from simple to challenging.

Comparison with Baselines. We compare `AutoEmbed` against three human-in-the-loop baselines: LLM-Prompt [26], requiring manual compiler info extraction for LLM debugging; Duinocode [18], an LLM-based Arduino code generator; and LLM-direct, direct LLM prompting for programming tasks. As shown in Tab. 6, `AutoEmbed` outperforms them with coding accuracy improvements of 32.4%, 15.6%, and 37.7%, and completion rate increases of 33.9%, 25.5%, and 53.4%. Duinocode achieves higher accuracy than others via partial library knowledge (e.g., examples). However, all baselines require human intervention, preventing automation.



(a) By module type. (b) By task difficulty.

Figure 12: Overall performance.

Table 6: Comparison with baselines.

System	Metrics		Features*		
	Cod. Acc.	Comp. Rate	Lib. Kno.	Debug	Auto.
LLM-Prompt [26]	72.3%	64.6%	○	◐	○
Duinocode [18]	82.8%	68.9%	◐	○	○
LLM-direct	69.5%	56.4%	○	○	○
`AutoEmbed`	95.7%	86.5%	●	●	●

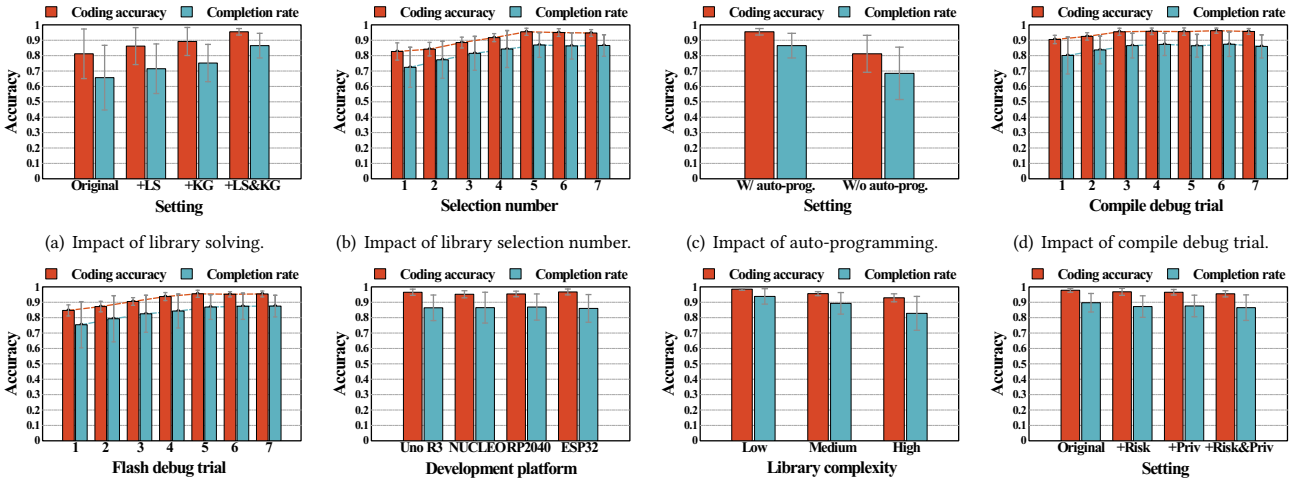
* ● applicable, ○ not applicable, ◐ partially applicable.

4.3 Micro-benchmark Evaluation

Library Solving and Knowledge Generation. The effectiveness of the library-solving approach is demonstrated in Fig. 13(a), where we compare the coding accuracy and completion rate of the proposed method against the use of the first search result. The results clearly show that `AutoEmbed` outperforms the approach without library solving, achieving a 10.8% increase in coding accuracy and a 21% improvement in the completion rate. We then evaluate the impact of the knowledge generation approach. As shown in Fig. 13(a), we compare the performance of the proposed method against using only the most similar example for the task. The results demonstrate that `AutoEmbed` outperforms the alternative, achieving a 7.1% increase in coding accuracy and a 15% improvement in completion rate. This finding highlights the effectiveness of our system in selecting and utilizing library knowledge.

Library Selection Number. `AutoEmbed` searches for libraries and retrieves the top N candidates for library selection as introduced in Sec. 3.1.2. Here, we evaluate the impact of N on performance. As shown in Fig. 13(b), the performance exhibited a positive correlation with the number of selected libraries. Increasing N from 1 to 3 resulted in a 7% improvement in coding accuracy and a 12.6% increase in completion rate. Further increasing N from 3 to 5 led to a 7.9% improvement in coding accuracy and a 6.6% increase in completion rate. This enhancement can be attributed to the fact that the most suitable library is not always among the top search results, but our selection algorithm can identify it when more options are considered. However, beyond a certain threshold, no further performance gains were observed. This is because expanding the range introduces more irrelevant libraries, which dilutes the effectiveness. Therefore, we choose 5 as the optimal number.

Auto-Programming. The impact of the auto-programming technique is evaluated by comparing performance metrics with and without the application of this method. As depicted in Fig. 13(c), the results demonstrate that `AutoEmbed` achieves a 17.6% increase in coding accuracy and a 26.2% improvement in completion rate. This finding highlights the effectiveness of `AutoEmbed` in automating the debugging process, leading to promising performance.



(a) Impact of library solving. (b) Impact of library selection number. (c) Impact of auto-programming. (d) Impact of compile debug trial.

(e) Impact of flash debug trial. (f) Impact of development platforms. (g) Impact of library complexity. (h) Impact of security checker.

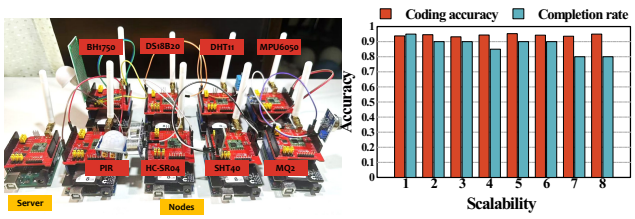
Figure 13: Experimental results.

Compile Debug Trials. In our investigation of the effect of compile debug trials on coding accuracy and completion rate within AutoEmbed, we observed notable trends, as illustrated in Fig. 13(d). Accuracy demonstrated a positive correlation with the number of trials: an increase from 1 to 3 trials led to a 5.5% improvement in coding accuracy and an 8% increase in completion rate. This enhancement can be attributed to the error information provided by the compiler, such as identifying library misuse, which facilitates the system's ability to correct these errors. However, beyond 3 trials, with up to 7 trials tested, no obvious improvements were observed. Therefore, we recommend maintaining a minimum of 3 compile debug trials to ensure reliable performance.

Flash Debug Trials. We then assessed the impact of flash debug trials on coding accuracy and completion rate within AutoEmbed, revealing significant trends, as depicted in Fig. 13(e). We observed a positive relationship between accuracy and the number of trials: an increase from 1 to 3 trials resulted in a 6.8% enhancement in coding accuracy and a 9.4% rise in completion rate. Furthermore, an increase from 3 to 5 trials led to a 5.5% improvement in coding accuracy and a 5.3% increase in completion rate. This improvement can be attributed to the debug information embedded during coding, which enhances the system's error correction capabilities. However, after conducting up to 7 trials, no significant enhancements were noted. Therefore, we recommend using 5 flash debug trials.

Development Platform. Then, we compared the performance of AutoEmbed across different development platforms. To ensure a fair comparison, we selected ten modules compatible with all four platforms (*i.e.*, Uno R3, NUCLEO-L4, Nano RP2040, and Nano ESP32). As shown in Fig. 13(f), all four platforms achieved over 95% coding accuracy and over 80% completion rates. These consistent results can be attributed to AutoEmbed's efficient automation design, which minimizes platform-specific dependencies and ensures a uniform development experience. This shows that AutoEmbed delivers consistently high performance on various platforms.

Library Complexity. We evaluate the impact of library complexity on AutoEmbed's performance by categorizing all modules based on the number of library APIs into three levels: low (fewer than 10 APIs), medium (10-20 APIs), and high (more than 20 APIs). As shown in Fig. 13(g), AutoEmbed achieved over 90% coding accuracy and over 80% completion rate across all levels of library complexity. These results indicate that AutoEmbed maintains consistent performance regardless of the complexity of the libraries involved.



(a) Setup. (b) Performance.

Figure 14: Scalability.

Security Checker. We evaluate AutoEmbed's performance after adding risk protection and privacy protection information to the prompt. As shown in Fig. 13(h), when risk protection was added, coding accuracy decreased by 1%, and completion rate decreased by 2.8%. When privacy protection was added, coding accuracy decreased by 1.3%, and completion rate decreased by 2.3%. When both protections were added simultaneously, there was a total decrease of 2.3% in coding accuracy and 3.6% in completion rate. However, the system's performance remained high.

Scalability. We evaluated the scalability of our system by developing embedded systems of varying scales. Specifically, we tested setups with 1 to 8 LoRa nodes, each transmitting sensor data to a LoRa server. Fig. 14(a) illustrates the LoRa devices used in the evaluation. As shown in Fig. 14(b), the coding accuracy remains stable. While the completion rate decreases as the scale approaches 8, it consistently stays above 80%, demonstrating robust scalability.

Overhead. We evaluate the token consumption and latency required by AutoEmbed to develop an embedded system with specific tasks. For this evaluation, we select 50 tasks from 10 modules that cover sensors, communication, display, and storage, sourced from EmbedTask. We calculate token consumption for each step using the usage data from API responses, and latency is measured using the built-in time calculation function in the debug tool. The token

Table 7: Overhead.

Meth. \ Ind. / Stg.	Token consumption (K)		Latency (s)	
	W/ Sel. Mem.	W/o Sel. Mem.	W/ Sel. Mem.	W/o Sel. Mem.
Lib. Solv.	-	-	4.9	5.2
Know. Gen.	7.925	7.893	95.7	89.8
Know. Extc.	0.224	0.232	2.3	1.8
Sec. Chk.	0.117	0.114	1.8	1.6
Cod. & Debug.	2.173	5.874	8.9	29.2
Total	10.439	14.113	113.6	127.6

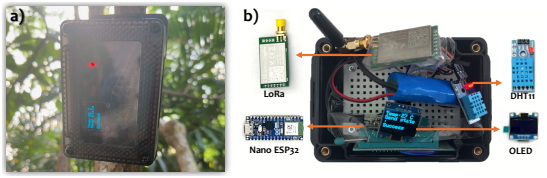


Figure 15: Environmental monitoring system deployment.

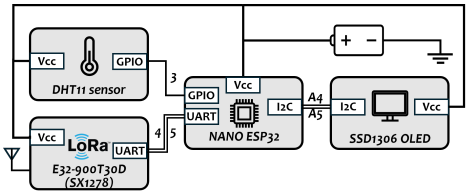


Figure 16: Environmental monitoring system architecture.

consumption and latency for different stages are presented in Tab. 7. AutoEmbed reduces runtime overhead by employing the selective memory pick-up method, as discussed in Sec. 3.3.1, which helps reduce prompt size. Therefore, we compare AutoEmbed with and without selective memory pick-up. By selectively picking relevant API information, AutoEmbed simplifies the necessary information, reducing token count by 26.2% (an average of 3.7K tokens) and latency by 11%. The benefits of this method include: (i) shorter token lengths decrease the inference latency of the LLM model, and (ii) reduced costs. For instance, with GPT-4o, the cost per 10 tasks decreases from $1.17 to $0.87 on average.

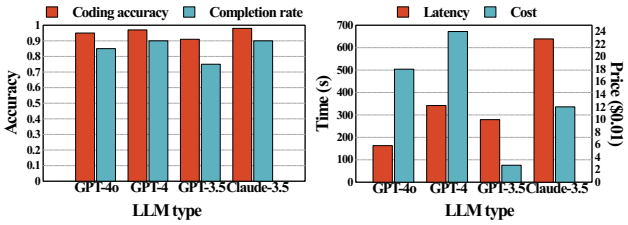
4.4 Case Study

4.4.1 Environmental Monitoring. This system combines a sensor for data collection, a transceiver for transmission, and a display for real-time updates.

Experimental Setup. The hardware architecture of the environmental monitoring system is depicted in Fig. 16. The Nano ESP32 board was selected as the development platform for its robust processing capabilities and compact size. A DHT11 sensor connects via pin 3 for accurate temperature readings, while an E32-900T30D LoRa module [67] on pins 4 and 5 enables long-range data transmission to a server. An SSD1306-based OLED display, connected via A4 (I2C SDA) and A5 (I2C SCL), provides real-time feedback.

Task Description. *"Develop an environmental monitor system that polls the DHT11 sensor every second to acquire temperature readings, transmits the temperature value to a server using the LoRa module, and updates the OLED display with the current temperature and LoRa transmission status."*

Metadata. *"Nano ESP32 connected with DHT11 to Pin 3, LoRa module to Pin 4 and 5, and OLED display to Pin A4 and A5."*



(a) Performance. (b) Overhead.

Figure 17: Case study 1 results.

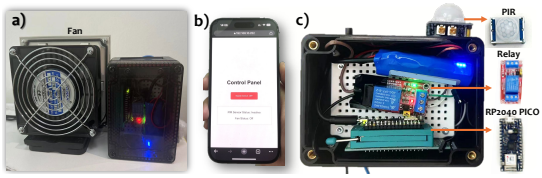


Figure 18: Remote control system deployment.

Performance. As shown in Fig. 15, AutoEmbed successfully developed the environmental monitoring system, which has been deployed in a forest for continuous environmental monitoring. We evaluate AutoEmbed on the first case study by comparing the coding accuracy and completion rate using different LLMs. The experiments involve running this task 20 times using AutoEmbed with each LLM. As shown in Fig. 17(a), GPT-4o, GPT-4, and Claude-3.5 achieve coding accuracy above 90% and completion rates exceeding 80%. In contrast, GPT-3.5 exhibits a lower performance, with a completion rate of around 70%. These results demonstrate that newer LLMs, such as GPT-4 and Claude-3.5, are better suited for complex coding tasks.

Overhead. We then evaluate the performance when using different LLMs to complete this task. As depicted in Fig. 17(b), Claude3.5 demonstrates the longest latency, exceeding 600 s, which is approximately four times longer than GPT-4o and two to three times longer than GPT-4 and GPT-3.5. Regarding cost efficiency, GPT-3.5 emerges as the most economical choice, thanks to its low unit pricing ($0.50 per 1M tokens for input and $1.50 per 1M tokens for output). To strike a balance between performance and overhead, we recommend opting for GPT-4o, as it offers lower latency.

4.4.2 Remote Control. This system demands not only embedded development but also front-end skills for web-based sensor status display and remote control.

Experimental Setup. As shown in Fig. 19, we utilized the Nano RP2040 development platform with the NORA-W106 Bluetooth Wi-Fi module as the embedded component. The HC-SR501 PIR sensor is responsible for detecting human presence, sending signals to pin 4 of the RP2040 PICO using high and low levels to indicate detection status. Based on the sensor input, the system controls a relay via pin 5 to manage the fan's connection to the AC power.

Task Description. *"Develop a remote control system that polls the PIR sensor every second, controls the fan via a relay, updates a web page with PIR status through the Wi-Fi module (Wi-Fi SSID: WiFiSSID, Password: PASSWORD), and includes a master switch button on the web to control system operation."*

Metadata. *"Nano RP2040 connected with PIR to Pin 4, relay to Pin 5, and WiFi module on-board."*

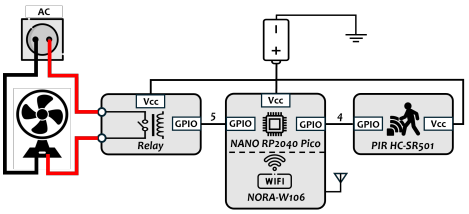
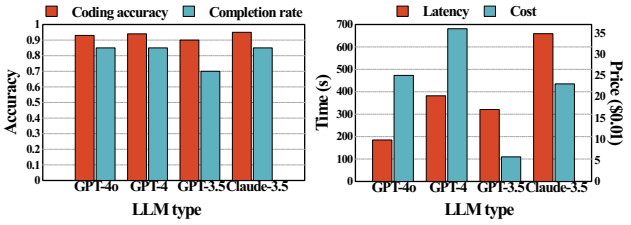


Figure 19: Remote control system architecture.



(a) Performance. (b) Overhead.

Figure 20: Case study 2 results.

Performance. As shown in Fig. 18, `AutoEmbed` successfully developed the remote fan control system, which has been deployed in a smart home to manage airflow based on occupancy automatically. Notably, in Fig.18(b), the system created a web interface that updates in real-time via Wi-Fi. We evaluate this case study by comparing the performance of `AutoEmbed` using different LLMs. As shown in Fig. 20(a), GPT-4o, GPT-4, and Claude 3.5 achieve completion rates exceeding 80%, while GPT-3.5 falls below 70%. This is likely due to the task's complexity, such as handling webpage setup through Wi-Fi, where GPT-3.5 may struggle with consistent accuracy. However, all four LLMs perform well overall, with coding accuracy exceeding 90%.

Overhead. We then assess the overhead of `AutoEmbed` when utilizing different LLMs to complete this task. As shown in Fig. 17(b), Claude 3.5 exhibits the longest latency, exceeding 600 s. While GPT-3.5 is theoretically faster, it requires more rounds of debugging, resulting in longer completion times than GPT-4o. In terms of cost, GPT-3.5 remains the most economical choice. Notably, GPT-4o excels in this task, offering low latency and strong performance, striking an ideal balance between speed and accuracy.

5 Related Work

LLM for Programming. Previous research has demonstrated that language models and AI assistants (*e.g.*, Github Copilot) can perform at the level of proficient software developers [12, 14, 23, 31, 35, 43, 47, 75, 90]. Researchers have also devised methods to prompt LLMs for code generation, including self-debugging [15, 29, 63, 89], task decomposition [39, 85], and prompting frameworks [54, 64]. However, existing studies lack guidance for specific embedded systems and do not cover domain knowledge crucial for embedded development.

LLM for Embedded System. Existing research on hardware-platform code generation mainly focuses on IDEs and programming frameworks without exploring LLMs in conjunction with hardware [7, 11, 22, 42, 51]. Other systems concentrate on HDL code for FPGAs [32, 44, 45]. Although some tools [18, 24, 56] support LLM-based embedded code generation, they primarily focus on basic code generation and fail to automate the full development process. Englhardt et al. [26] assess LLMs for embedded development but focus on human-in-the-loop tests, lacking automation.

LLM in IoT. Recent research has focused on leveraging sensor data to enhance LLMs in understanding the physical world [16, 20, 27, 34, 36, 38, 40, 50, 52, 60, 61, 71, 76, 79]. For instance, BABEL [20] aligns multiple sensing modalities to bridge the gap between LLMs and sensory comprehension. LLMs are also adapted for mobile devices [73, 74, 86–88], *e.g.*, AutoDroid [73] utilizes LLMs to control Android apps. An et al. [3] introduce IoT-LLM to integrate IoT sensor data into LLMs for physical-world reasoning.

6 Discussion

Handling Imperfect Library Information. Embedded systems rely on third-party libraries that are often poorly documented or unmaintained. As described in Sec. 3.1.2, AutoEmbed addresses this through a scoring-based library solving method that prioritizes well-maintained, compatible libraries, minimizing missing library issues across over 7,000 libraries.

Hardware Failures Handling. Embedded systems are prone to hardware failures, especially non-deterministic ones like intermittent I2C or SPI errors, which are difficult to diagnose and limit software automation. While AutoEmbed automates the software development pipeline, hardware issues, particularly those affected by environmental factors, necessitate manual intervention as noted by Koca et al. [41]. For non-deterministic failures, users can perform manual hardware checks. Integrating automated diagnostic capabilities is a promising direction for future work.

Scalability of Pin Specification. The current system requires manual input of pin connections, which may not scale effectively to complex multi-peripheral boards or production-level embedded systems. Future work could explore automated pin assignment based on board layouts and component requirements.

Sequential Task Focus. AutoEmbed is primarily evaluated on sequential tasks and may not handle concurrent, distributed, or real-time scenarios effectively. Extending the system to support RTOS and concurrent task execution is an important future direction.

Knowledge Base Quality Dependency. The system's performance relies on the quality of generated library knowledge. Incomplete or inaccurate library information can impact performance, though the auto-programming feedback loop mitigates this through iterative error correction.

Platform Extensibility. While evaluated on four platforms, the framework is designed to be extensible. Adding support for new platforms requires providing the corresponding toolchain and board information, which is a straightforward but manual process.

LLM Performance Analysis. Our analysis reveals significant performance differences among LLM models. GPT-3.5 achieves notably lower completion rates compared to GPT-4o and Claude 3.5, as its weaker reasoning capabilities are inadequate for handling structured API knowledge and complex embedded programming logic. More advanced models demonstrate superior capability in managing hardware dependencies and multi-step development processes.

7 Conclusion

This study takes the first step toward automating embedded IoT system development with no manual intervention. AutoEmbed introduces a library resolution method, a library knowledge generation method, and an auto-programming method. Extensive evaluation across 71 hardware modules, four platforms, and over 350 IoT tasks shows that AutoEmbed achieves an average completion rate of 86.5%. We will release AutoEmbed as an open-source tool and future work will focus on extending it to more complex embedded IoT systems.

Acknowledgments

This project was supported by National Key R&D Program of China (Grant No. 2023YFE0208800), the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11202925, CityU 11201422, and CityU 11205624), NSF of Guangdong Province (Project No. 2024A1515010192), and the CityU REG-Postdoc Fellow Fund (Project No. 2024PF061).

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Akiko Aizawa. 2003. An information-theoretic perspective of tf–idf measures. *Information Processing & Management* 39, 1 (2003), 45–65.

[3] Tuo An, Yunjiao Zhou, Han Zou, and Jianfei Yang. 2024. Iot-llm: Enhancing real-world iot task reasoning with large language models. *arXiv preprint arXiv:2410.02429* (2024).

[4] Arduino. 2024. Adafruit SSD1306 OLED Library. https://www.arduino.cc/reference/en/libraries/adafruit-ssd1306/. Accessed: 2024-08-30.

[5] Arduino. 2024. Arduino Libraries. https://www.arduino.cc/en/Reference/Libraries. Accessed: 2024-08-22.

[6] Arduino. 2024. FastLED Library. https://www.arduino.cc/reference/en/libraries/fastled/. Accessed: 2024-08-30.

[7] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michał Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *ACM SPLASH*.

[8] Michael Barr and Anthony Massa. 2006. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, Inc.

[9] Julien Bayle. 2013. *C programming for Arduino*. Packt Publishing Ltd.

[10] Barry W Boehm. 1984. Software engineering economics. *IEEE TSE* SE-10, 1 (1984), 4–21.

[11] Robert W Brennan and Jonathan Lesage. 2022. Exploring the implications of OpenAI codex on education for industry 4.0. In *Springer SOHOMA*.

[12] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[13] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *ICLR*.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[16] Xingyu Chen and Xinyu Zhang. 2023. Rf genesis: Zero-shot generalization of mmwave sensing through simulation-based data synthesis and generative diffusion models. In *ACM SenSys*.

[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *JMLR* 24 (2023), 1–113.

[18] CJS Robotics. 2024. Duino Code Generator. https://www.duinocodegenerator.com/ Accessed: 2024-08-20.

[19] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

[20] Shenghong Dai, Shiqi Jiang, Yifan Yang, Ting Cao, Mo Li, Suman Banerjee, and Lili Qiu. 2024. Advancing Multi-Modal Sensing Through Expandable Modality Alignment. *arXiv preprint arXiv:2407.17777* (2024).

[21] Tuan Dang, Trung Tran, Khang Nguyen, Tien Pham, Nhat Pham, Tam Vu, and Phuc Nguyen. 2022. ioTree: a battery-free wearable system with biocompatible sensors for continuous tree health monitoring. In *ACM MobiCom*.

[22] James Devine, Joe Finney, Peli De Halleux, Michał Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. *ACM SIGPLAN Notices* 53, 4 (2018), 19–30.

[23] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *PMLR ICML*.

[24] DroneBot Workshop. 2023. Using ChatGPT to Write Code for Arduino and ESP32. https://dronebotworkshop.com/chatgpt/ Accessed: 2024-08-20.

[25] Di Duan, Huanqi Yang, Guohao Lan, Tianxing Li, Xiaohua Jia, and Weitao Xu. 2023. EMGSense: A Low-Effort Self-Supervised Domain Adaptation Framework for EMG Sensing. In *IEEE PerCom*. IEEE, 160–170. doi:10.1109/PERCOM56429.2023.10099164

[26] Zachary Englhardt, Richard Li, Dilini Nissanka, Zhihan Zhang, Girish Narayanswamy, Joseph Breda, Xin Liu, Shwetak Patel, and Vikram Iyer. 2024. Exploring and characterizing large language models for embedded system development and debugging. In *ACM CHI EA*.

[27] Zachary Englhardt, Chengqian Ma, Margaret E Morris, Chun-Cheng Chang, Xuhai Xu, Lianhui Qin, Daniel McDuff, Xin Liu, Shwetak Patel, and Vikram Iyer. 2024. From Classification to Clinical Insights: Towards Analyzing and Reasoning About Mobile and Behavioral Health Data With Large Language Models. *ACM IMWUT* 8, 2 (2024), 1–25.

[28] Brian Evans. 2011. *Beginning arduino programming*. Apress.

[29] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madan Musuvathi, and Shuvendu Lahiri. 2024. Exploring the Effectiveness of LLM based Test-driven Interactive Code Generation: User Study and Empirical Evaluation. In *IEEE/ACM ICSE*.

[30] Kaijie Gong, Wei Dong, Yingqi Peng, Hao Wang, and Yi Gao. 2024. Poster: Enabling IoT Application Programming in Natural Language with IoTPilot. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. 903–904.

[31] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644* (2023).

[32] Zhi Guo, Walid Najjar, and Betul Buyukkurt. 2008. Efficient hardware code generation for FPGAs. *ACM TACO* 5, 1 (2008), 1–26.

[33] Mingda Han, Huanqi Yang, Tao Ni, Di Duan, Mengzhe Ruan, Yongliang Chen, Jia Zhang, and Weitao Xu. 2024. mmSign: mmWave-based Few-Shot Online Handwritten Signature Verification. *ACM Transactions on Sensor Networks* 20, 4 (2024), 89:1–89:31. doi:10.1145/3605945

[34] Aritra Hota, Soumyajit Chatterjee, and Sandip Chakraborty. 2024. Evaluating Large Language Models as Virtual Annotators for Time-series Physical Sensing Data. *arXiv preprint arXiv:2403.01133* (2024).

[35] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *IEEE/ACM ICSE*.

[36] Sijie Ji, Xinzhe Zheng, and Chenshu Wu. 2024. HARGPT: Are LLMs Zero-Shot Human Activity Recognizers? *arXiv preprint arXiv:2403.02727* (2024).

[37] Manuel Jiménez, Rogelio Palomera, and Isidoro Couvertier. 2013. *Introduction to embedded systems*. Springer.

[38] Ming Jin, Yifan Zhang, Wei Chen, Kexin Zhang, Yuxuan Liang, Bin Yang, Jindong Wang, Shirui Pan, and Qingsong Wen. 2024. Position: What Can Large Language Models Tell Us about Time Series Analysis. In *ICML*.

[39] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. In *NeurIPS*.

[40] Yubin Kim, Xuhai Xu, Daniel McDuff, Cynthia Breazeal, and Hae Won Park. 2024. Health-llm: Large language models for health prediction via wearable sensor data. *arXiv preprint arXiv:2401.06866* (2024).

[41] Aylin Koca, Mathias Funk, Evangelos Karapanos, Anne Rozinat, Wil M. P. Van Der Aalst, Henk Corporaal, Jean-Bernard O. S. Martens, Piet H. A. Van Der Putten, A. J. M. M. Weijters, and Aarnout Cornelis Brombacher. 2009. Soft Reliability: an interdisciplinary approach with a user–system focus. *Quality and reliability engineering international* 25, 1 (2009), 3–20.

[42] Philip Koopman, Howie Choset, Rajeev Gandhi, Bruce Krogh, Diana Marculescu, Priya Narasimhan, Joann M Paul, Ragunathan Rajkumar, Daniel Siewiorek, Asim Smailagic, et al. 2005. Undergraduate embedded system education at Carnegie Mellon. *ACM TECS* 4, 3 (2005), 500–528.

[43] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *NeurIPS*.

[44] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. 2016. Automatic code generation of convolutional neural networks in FPGA implementation. In *IEEE*

FPT.
[45] Tomas G Moreira, Marco A Wehrmeister, Carlos E Pereira, Jean-Francois Petin, and Eric Levrat. 2010. Automatic code generation for embedded systems: From UML specifications to VHDL code. In *IEEE INDIN*.
[46] S. Neelakandan, M. A. Berlin, Sandesh Tripathi, V. Brindha Devi, Indu Bhardwaj, and N. Arulkumar. 2021. IoT-based traffic prediction and traffic signal control system for smart city. *Soft Computing* 25 (2021), 12241–12248.
[47] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *MSR*.
[48] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *ICML*.
[49] OpenAI. 2022. ChatGPT. https://openai.com/blog/chatgpt/. Accessed: 2024-08-22.
[50] Xiaomin Ouyang and Mani Srivastava. 2024. LLMSense: Harnessing LLMs for High-level Reasoning Over Spatiotemporal Sensor Traces. *arXiv preprint arXiv:2403.19857* (2024).
[51] Sudeep Pasricha. 2022. Embedded systems education in the 2020s: Challenges, reflections, and future directions. In *GLSVLSI*.
[52] Danilo Pietro Pau and Fabrizio Maria Aymone. 2024. Forward learning of large language models by consumer devices. *Electronics* 13, 2 (2024), 402.
[53] Nhat Pham, Hong Jia, Minh Tran, Tuan Dinh, Nam Bui, Young Kwon, Dong Ma, Phuc Nguyen, Cecilia Mascolo, and Tam Vu. 2022. PROS: an efficient pattern-driven compressive sensing framework for low-power biopotential-based wearables with on-chip intelligence. In *ACM MobiCom*.
[54] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *ICLR*.
[55] Precedence Research. 2024. Embedded Systems Market Size, Share, Trends, Report 2023 to 2032. https://www.precedenceresearch.com/embedded-systems-market. Accessed: 2024-09-03.
[56] Puneeth. 2023. I tried ChatGPT for Arduino - It's Surprising. https://blog.wokwi.com/learn-arduino-using-ai-chatgpt/ Accessed: 2024-08-20.
[57] M Revathy, S Ramya, R Sathiyavathi, B Bharathi, and V Maria Anu. 2017. Automation of street light for smart city. In *IEEE ICCSP*.
[58] SaM Solutions. 2024. All You Need to Know About Embedded System Programming. https://www.sam-solutions.com/blog/all-you-need-to-know-about-embedded-system-programming/ Accessed: 2024-07-29.
[59] Pedro M Santos, João GP Rodrigues, Susana B Cruz, Tiago Lourenço, Pedro M d'Orey, Yunior Luis, Cecília Rocha, Sofia Sousa, Sérgio Crisóstomo, Cristina Queirós, et al. 2018. PortoLivingLab: An IoT-based sensing platform for smart cities. *IEEE IoTJ* 5, 2 (2018), 523–532.
[60] Leming Shen, Qiang Yang, Xinyu Huang, Zijing Ma, and Yuanqing Zheng. 2025. GPIoT: Tailoring Small Language Models for IoT Program Synthesis and Development. In *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*. 199–212.
[61] 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* (2025).
[62] Shuyao Shi, Neiwen Ling, Zhehao Jiang, Xuan Huang, Yuze He, Xiaoguang Zhao, Bufang Yang, Chen Bian, Jingfei Xia, Zhenyu Yan, et al. 2024. Soar: Design and Deployment of A Smart Roadside Infrastructure System for Autonomous Driving. In *ACM MobiCom*.
[63] Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366* (2023).
[64] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *PMLR ICML*.
[65] Zehua Sun, Tao Ni, Huanqi Yang, Kai Liu, Yu Zhang, Tao Gu, and Weitao Xu. 2023. FLoRa: Energy-Efficient, Reliable, and Beamforming-Assisted Over-The-Air Firmware Update in LoRa Networks. In *ACM/IEEE IPSN*. ACM, 14–26. doi:10.1145/3583120.3586963
[66] Zehua Sun, Tao Ni, Huanqi Yang, Kai Liu, Yu Zhang, Tao Gu, and Weitao Xu. 2024. FLoRa+: Energy-efficient, Reliable, Beamforming-assisted, and Secure Over-the-air Firmware Update in LoRa Networks. *ACM Transactions on Sensor Networks* 20, 3 (2024), 54:1–54:28. doi:10.1145/3641548
[67] Zehua Sun, Huanqi Yang, Kai Liu, Zhimeng Yin, Zhenjiang Li, and Weitao Xu. 2022. Recent Advances in LoRa: A Comprehensive Survey. *ACM Transactions on Sensor Networks* 18, 4 (2022), 67:1–67:44. doi:10.1145/3543856
[68] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[69] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. 2017. FarmBeats: an IoT platform for Data-Driven agriculture. In *USENIX NSDI*.
[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*.
[71] Qijun Wang, Shichen Zhang, Kunzhe Song, and Huacheng Zeng. 2024. ChatTracer: Large Language Model Powered Real-time Bluetooth Device Tracking System. *arXiv preprint arXiv:2403.19833* (2024).
[72] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*.
[73] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *ACM MobiCom*.
[74] Daliang Xu, Wangsong Yin, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. 2023. Llmcad: Fast and scalable on-device large language model inference. *arXiv preprint arXiv:2309.04255* (2023).
[75] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *ACM MAPS*.
[76] Huatao Xu, Liying Han, Qirui Yang, Mo Li, and Mani Srivastava. 2024. Penetrative ai: Making llms comprehend the physical world. In *HotMobile*. 1–7.
[77] Ruiyang Xu, Jialun Cao, Mingyuan Wu, Wenliang Zhong, Yaojie Lu, Ben He, Xianpei Han, Shing-Chi Cheung, and Le Sun. 2025. EmbedAgent: Benchmarking Large Language Models in Embedded System Development. *arXiv preprint arXiv:2506.11003* (2025).
[78] Weitao Xu, Huanqi Yang, Jiongzhang Chen, Chengwen Luo, Jia Zhang, Yuliang Zhao, and Wen Jung Li. 2024. WashRing: An Energy-Efficient and Highly Accurate Handwashing Monitoring System Via Smart Ring. *IEEE Transactions on Mobile Computing* 23, 1 (2024), 971–984. doi:10.1109/TMC.2022.3227299
[79] Bufang Yang, Siyang Jiang, Lilin Xu, Kaiwei Liu, Hai Li, Guoliang Xing, Hongkai Chen, Xiaofan Jiang, and Zhenyu Yan. 2024. DrHouse: An LLM-empowered Diagnostic Reasoning System through Harnessing Outcomes from Sensor Data and Expert Knowledge. *arXiv preprint arXiv:2405.12541* (2024).
[80] Huanqi Yang, Di Duan, Hongbo Liu, Chengwen Luo, Yuezhong Wu, Wei Li, Albert Y. Zomaya, Linqi Song, and Weitao Xu. 2024. Scenario-Adaptive Key Establishment Scheme for LoRa-Enabled IoV Communications. *IEEE Transactions on Mobile Computing* 23, 12 (2024), 12998–13014. doi:10.1109/TMC.2024.3421659
[81] Huanqi Yang, Mingda Han, Mingda Jia, Zehua Sun, Pengfei Hu, Yu Zhang, Tao Gu, and Weitao Xu. 2023. XGait: Cross-Modal Translation via Deep Generative Sensing for RF-based Gait Recognition. In *ACM SenSys*. ACM, 43–55. doi:10.1145/3625687.3625792
[82] Huanqi Yang, Mingda Han, Xinyue Li, Di Duan, Tianxing Li, and Weitao Xu. 2025. iRadar: Synthesizing Millimeter-Waves from Wearable Inertial Inputs for Human Gesture Sensing. In *IEEE INFOCOM*. IEEE, 1–10. doi:10.1109/INFOCOM55648.2025.11044481
[83] Huanqi Yang, Mingda Han, Shuyao Shi, Zhenyu Yan, Guoliang Xing, Jianping Wang, and Weitao Xu. 2023. Wave-for-Safe: Multisensor-based Mutual Authentication for Unmanned Delivery Vehicle Services. In *ACM MobiHoc*. ACM, 230–239. doi:10.1145/3565287.3610253
[84] Huanqi Yang, Zehua Sun, Hongbo Liu, Xianjin Xia, Yu Zhang, Tao Gu, Gerhard P. Hancke, and Weitao Xu. 2023. ChirpKey: A Chirp-Level Information-based Key Generation Scheme for LoRa Networks via Perturbed Compressed Sensing. In *IEEE INFOCOM*. IEEE, 1–10. doi:10.1109/INFOCOM53939.2023.10228886
[85] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*.
[86] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2023. Edgemoe: Fast on-device inference of moe-based large language models. *arXiv preprint arXiv:2308.14352* (2023).
[87] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. 2024. Llm as a system service on mobile devices. *arXiv preprint arXiv:2403.11805* (2024).
[88] Li Zhang, Shihe Wang, Xianqing Jia, Zhihan Zheng, Yunhe Yan, Longxi Gao, Yuanchun Li, and Mengwei Xu. 2024. LlamaTouch: A Faithful and Scalable Testbed for Mobile UI Task Automation. In *ACM UIST*.
[89] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. LDB: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step. In *ACL: Findings*.
[90] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *ACM MAPS*.