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

Huanqi Yang[1], Mingzhe Li[1], Mingda Han[2], Zhenjiang Li[1], Weitao Xu[1]
[1]City University of Hong Kong, [2]Shandong University

Background

Embedded IoT Systems: A Growing Market

- Market projected to reach **$258.6B by 2032**
- Deployed across healthcare, agriculture, smart cities, and human sensing
- E.g., smart street lighting, traffic signal control, environmental monitoring

Traditional Development Pipeline

Core Problem

Embedded system development straddles the **hardware–software interface**, demanding cross-domain expertise to manage device–physical-world interactions.

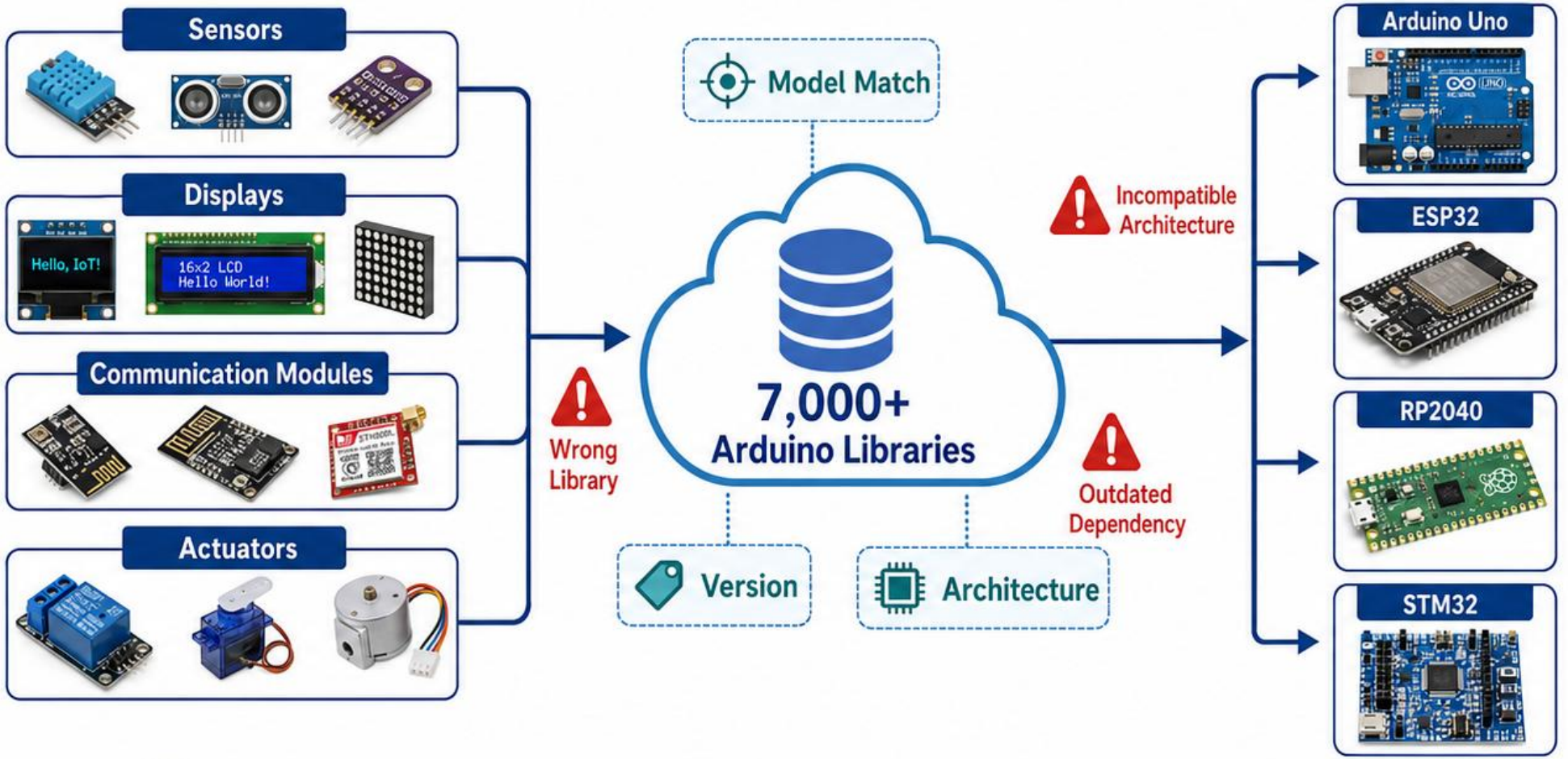
Traditional Process is...

- **Labor-intensive** — manual dependency resolution across 7,000+ libraries
- **Time-consuming** — iterative compile–flash–debug cycles
- **Error-prone** — diverse hardware, IDEs (Eclipse, Keil, IAR), and MCU architectures

Research Gap: Existing LLM coding tools (CodeT, LDB, Copilot) lack *hardware-specific knowledge* required for embedded development.

Challenge 1: Diversity in Hardware Dependency

Selecting the right library is hard in a heterogeneous IoT ecosystem.

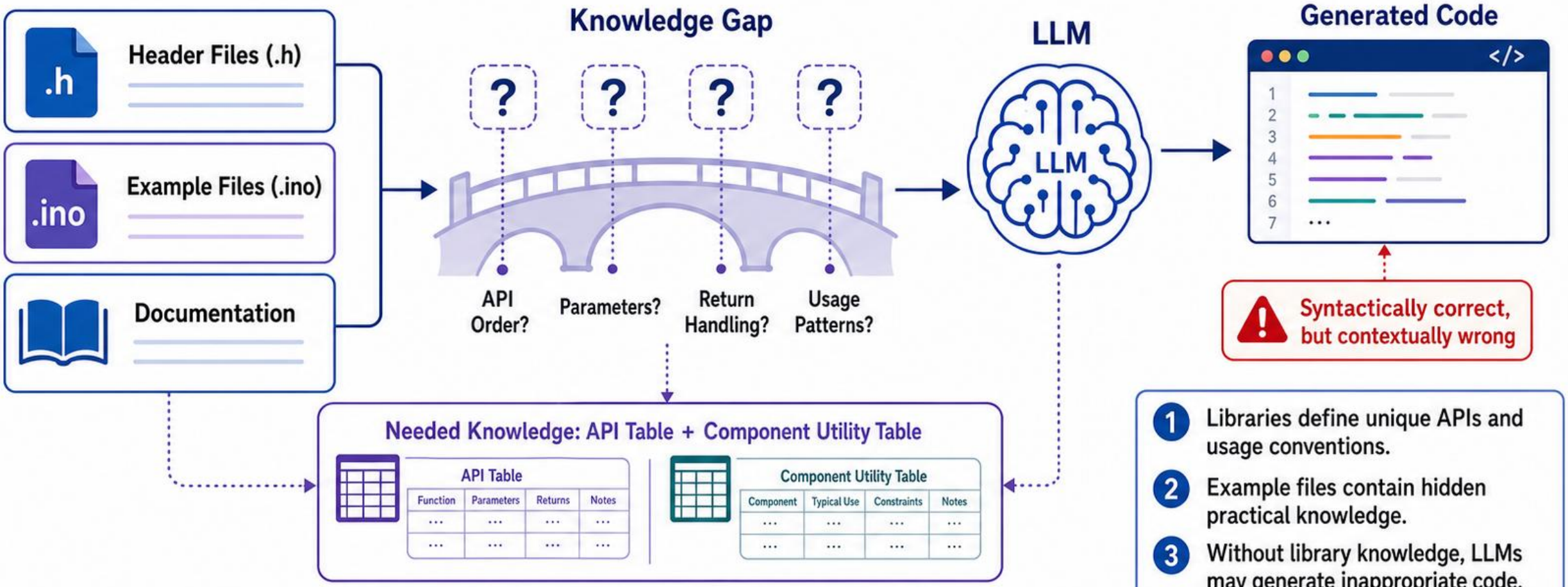


1. Hardware modules differ in architecture and function.
2. Each component requires specific library dependencies.
3. Manual dependency search is labor-intensive and error-prone.

Key barrier: accurate, platform-compatible dependency resolution.

Challenge 2: Lack of Library Knowledge

Generic LLMs lack the specialized API semantics needed for embedded coding.

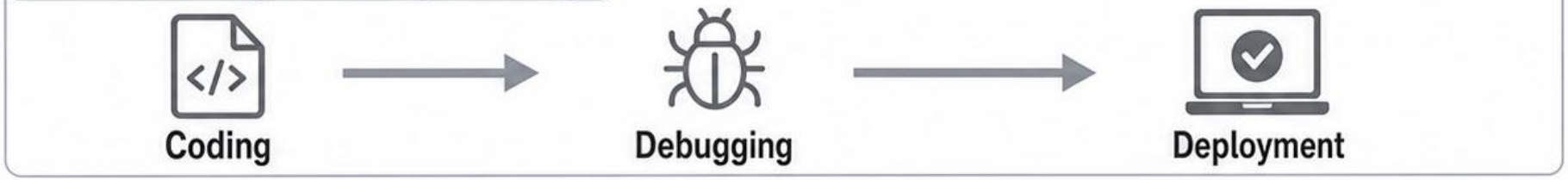


Key barrier: injecting structured API and utility knowledge into LLM memory.

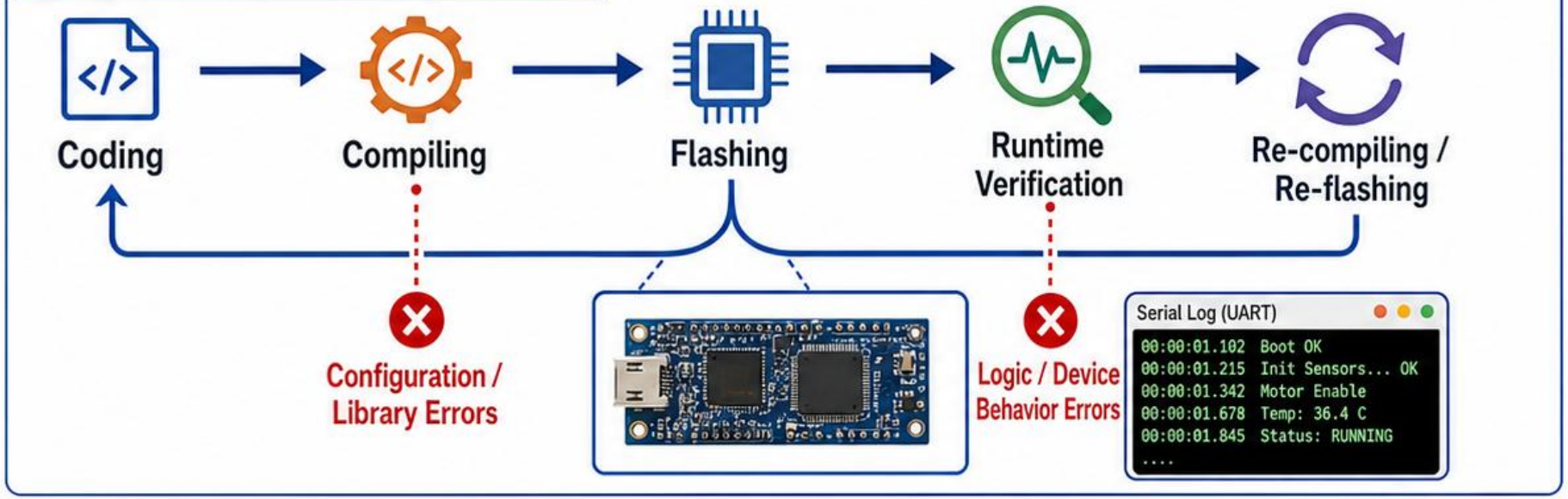
Challenge 3: Complexity of Embedded System Programming

Embedded development requires hardware-in-the-loop validiation.

General-Purpose Programming



Embedded System Programming

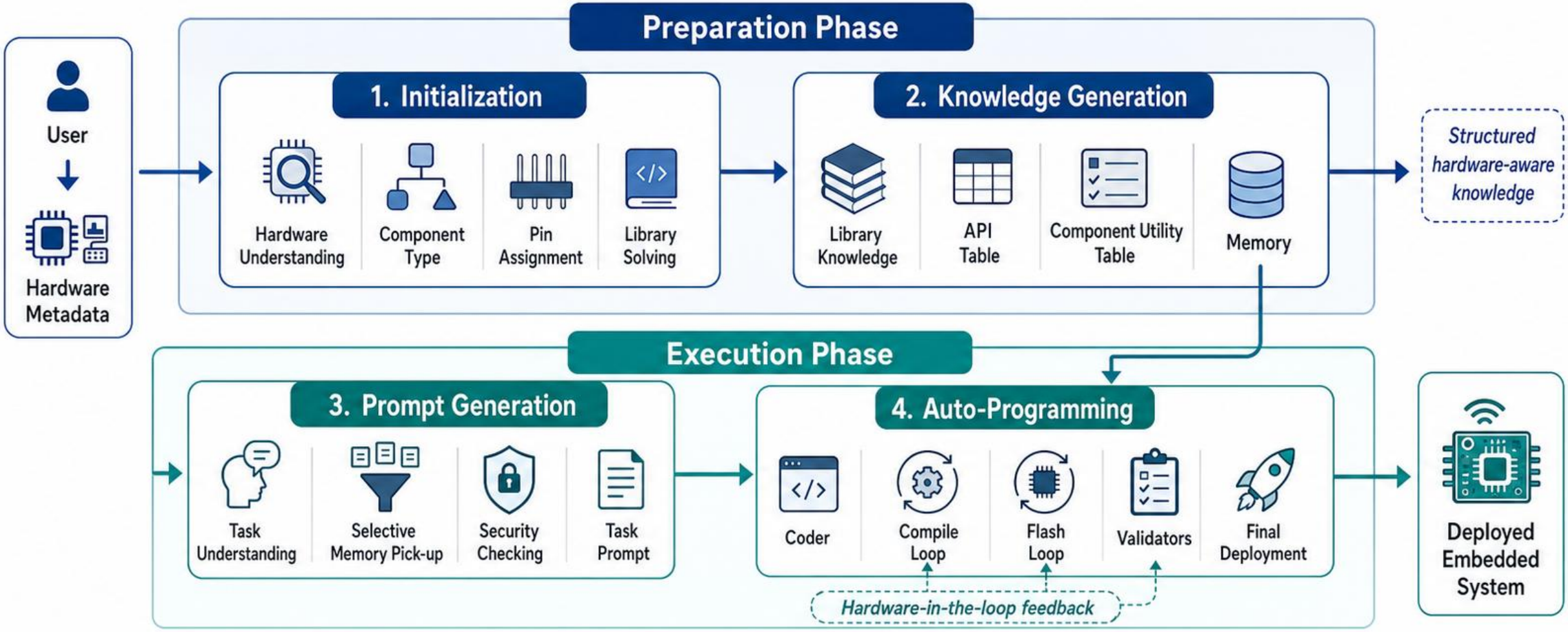


Key Points

1. Code must satisfy both software syntax and hardware configuration.
2. Errors appear during compilation and after flashing.
3. Functional correctness depends on observed device behavior.

Key barrier: automated feedback loops for compile-time and runtime correction.

System Design Overview: From User Task to Deployed IoT System

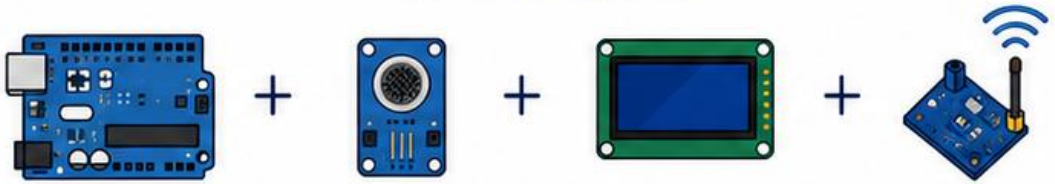


AutoEmbed decomposes embedded development into **dependency solving**, **knowledge injection**, **prompt construction**, and **validated deployment**.

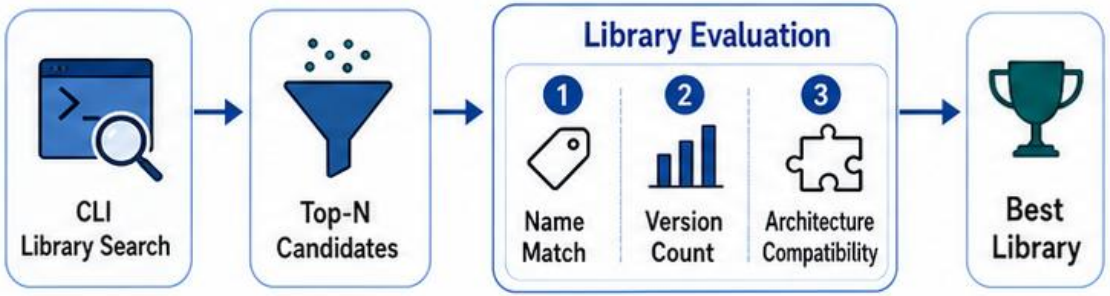
Preparation Phase: Hardware-Aware Library Resolution and Knowledge Generation

Step 1: Initialization & Library Solving

Hardware Context

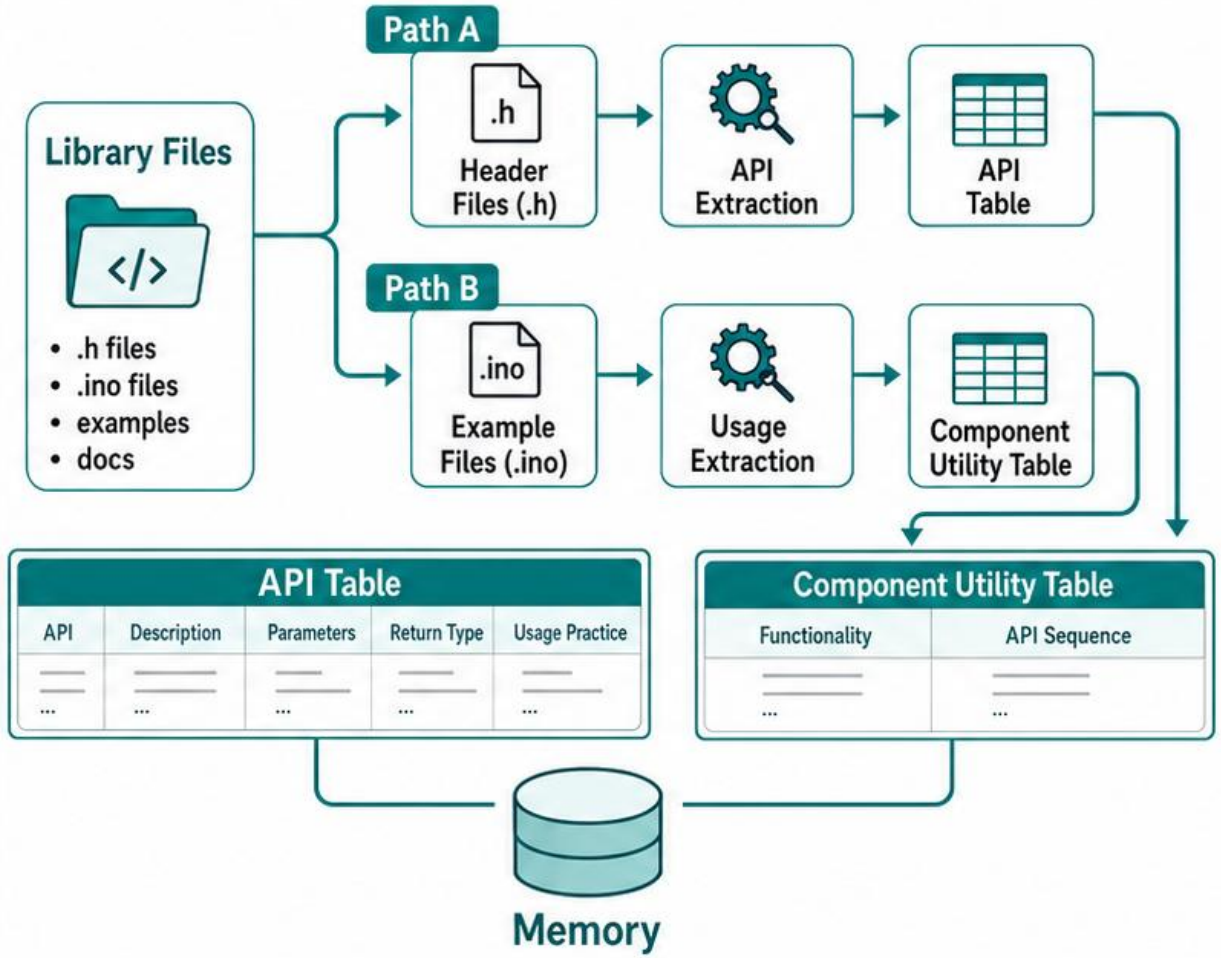


Hardware-Aware Library Resolution



Selects compatible and actively maintained libraries automatically.

Step 2: Knowledge Generation



Preparation transforms raw hardware libraries into structured knowledge that LLMs can reliably use.

Execution Phase: Selective Prompting and Auto-Programming

1. User Task → **2. Functionality Separation** → **3. Selective Memory Pick-up** → **4. Structured Task Prompt** → **5. Auto-Programming**

User Task

Read sensor data, transmit it, and update display

Functionality Separation

- Read Sensor
- Transmit Data
- Update Display

Selective Memory Pick-up

Utility Table Matching

API Table Lookup

Only task-relevant APIs are inserted into the prompt

Structured Task Prompt

Task
High-level task description and objectives

Hardware Metadata
Board, components, pins, interfaces, constraints

API Usage
Only task-relevant APIs with parameters and usage hints

Coding Rules
Language, style, error handling, resource and memory rules

Auto-Programming

Loop 1: Compile Loop

Coder → Compiler → Compile Validator

Fix syntax, library, and configuration errors

Loop 2: Flash Loop

Coder → Flash to Board → Runtime Logs → Flash Validator

Check order, logic, and real hardware behavior

DEBUG logs guide validation
Provide traces and diagnostics to accelerate issue resolution.

Security Checking

Risk Protection
Detect unsafe code patterns and malicious behavior.

Privacy Masking
Mask sensitive data and protect user privacy.

AutoEmbed closes the loop between LLM-generated code and real embedded hardware until deployment succeeds.

Evaluation: Broad Hardware Coverage and Strong Performance

4 Platforms
Uno R3, NUCLEO-L4, Nano RP2040, Nano ESP32

71 Modules
Sensors, communication, displays, control devices

355 IoT Tasks
Level 1 to Level 3 complexity

2 Metrics
Coding Accuracy + Completion Rate

95.7%
Average Coding Accuracy

86.5%
Average Completion Rate

Method	Coding Accuracy	Completion Rate
LLM-direct	69.5%	56.4%
LLM-Prompt	72.3%	64.6%
Duinocode	82.8%	68.9%
AutoEmbed	95.7%	86.5%

Best Performance

AutoEmbed outperforms human-in-the-loop LLM baselines by automating library knowledge, debugging, and deployment.

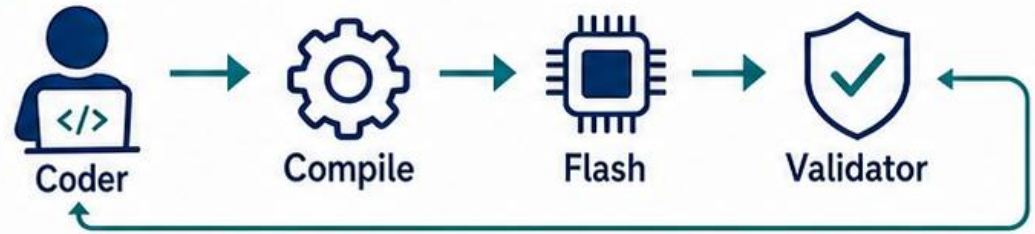
Micro-Benchmark Insights: What Drives AutoEmbed's Performance?

1 Library Solving + Knowledge Generation

+10.8% Coding Accuracy
+21.0% Completion Rate
Library solving vs. first search result

+7.1% Coding Accuracy
+15.0% Completion Rate
Knowledge generation vs. single example

2 Auto-Programming Feedback



+17.6% Coding Accuracy
+26.2% Completion Rate
Compile loop + flash loop detect and correct errors

3 Robust Across Hardware

>95%
Coding Accuracy across platforms

>80%
Completion Rate across platforms

>90%
Coding Accuracy across library complexity levels

4 Efficient and Scalable

-26.2%
Tokens

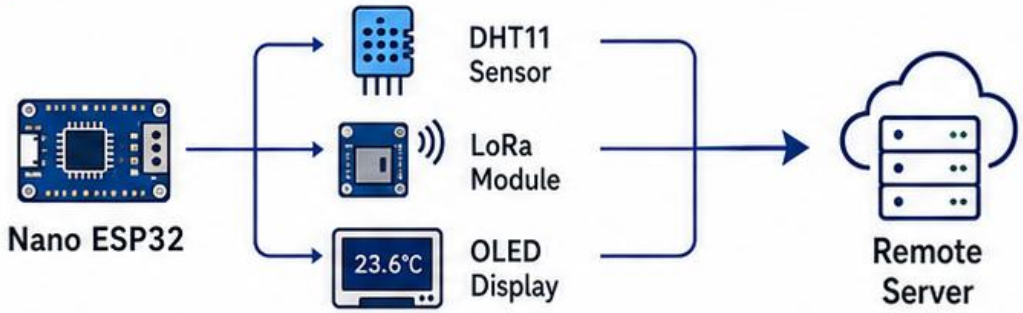
-11%
Latency

>80%
Completion Rate remains >80% with 1–8 LoRa nodes

Recommended settings: Top-5 library candidates, 3 compile trials, 5 flash trials.

Case Studies: From Natural Task Description to Working IoT Systems

Case Study 1: Environmental Monitoring



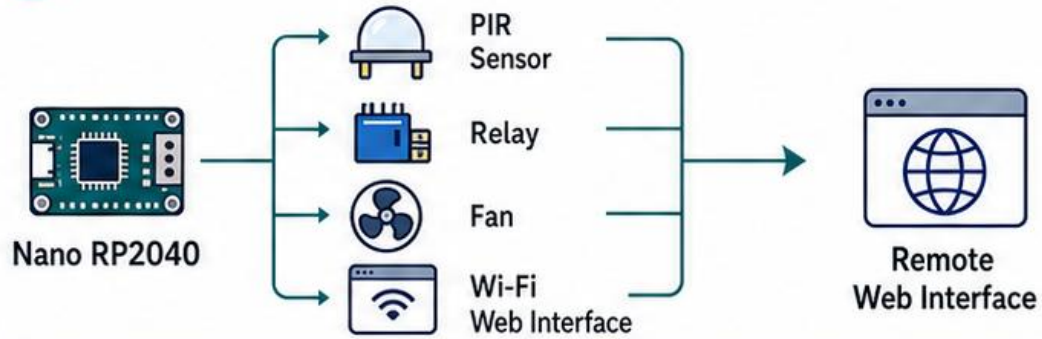
Tasks

- Poll temperature every second
- Transmit data via LoRa
- Display temperature and transmission status

Developed in 2.6 min

Deployment
Forest environmental monitoring

Case Study 2: Remote Control System



Tasks

- Detect human presence
- Control fan via relay
- Update web page in real time
- Master switch on web UI

Developed in 3.1 min

Deployment
Smart home airflow control

1. Automates dependency solving

Resolves libraries, versions, and compatibility to eliminate manual trial and error.

2. Injects structured library knowledge

Leverages a curated knowledge base to select the right components and APIs.

3. Validates code through compile + flash feedback

Uses real compile and flash results to detect and correct issues automatically.

AutoEmbed lowers the barrier to reliable embedded IoT development.

Conclusion

1. AutoEmbed automates the full embedded IoT development pipeline. It reduces manual effort by automatically handling hardware dependency solving, code generation, compilation, flashing, debugging, and final deployment.

2. It combines component-aware library resolution, structured library knowledge generation, selective prompt construction, and compile/flash feedback loops to make LLM-generated code hardware-aware and deployable. It demonstrates strong effectiveness and practical potential.

3. Open-source project website: https://autoembed.github.io/