Rebuttal for ICSE 2026 Submission #50

Reviewer #50A

Q1: Clarity on handling imperfect library information in Library Resolution. A1: As detailed in Sec 3.1.2 and Alg. 1, AutoEmbed's library solving method scores libraries on name similarity, version count (favoring active maintenance), and architecture compatibility. This implicitly penalizes poorly documented or unmaintained libraries, which might yield parsing issues or low version counts. AutoEmbed currently focuses on selecting the most suitable library from available, maintained, and compatible options. Explicitly addressing scenarios with severely misconfigured or entirely missing documentation remains an area for future work.

Q2: Handling of non-deterministic hardware failures in Auto-Programming (Sec 3.4). A2: Non-deterministic hardware failures (e.g., intermittent I2C/SPI issues) are not primarily software engineering (SE) problems; diagnosing them is challenging and time-consuming, even for specialists with hardware tools. AutoEmbed's auto-programming (Sec 3.4) targets software-detectable functional and logical errors via DEBUG INFO and Order/Logic Checks. Deep hardware debugging is beyond our current SE automation scope.

Q3: Regarding security and safety. A3: AutoEmbed currently prioritizes functional programming for embedded systems. A security check module was developed but omitted due to page limits; we plan to integrate and detail it in the final version.

Q4: Definition of 'coding accuracy' and 'completion rate' regarding I/O, timing, or power states. A4: Sec 4.1 defines 'Coding Accuracy' as correct API usage and 'Completion Rate' as one-attempt task success. While these metrics focus on API-level and task success, Auto-Programming's Flash Loop validator (Fig. 5) uses logs to check I/O correctness and execution sequence. For example, its Logic Check verifies if peripheral states change correctly based on inputs (Fig. 9). Successful task completion thus implies these I/O and timing aspects were sufficiently addressed. Fig. 13(c) demonstrates that without Auto-Programming's I/O correctness checks, the completion rate decreases.

Q5: Evaluation of system-level properties (flash/RAM usage, power). A5: While Table 1 provides platform specifications (FLASH, SRAM), a detailed analysis of resource utilization (flash/RAM footprint, power consumption) for the generated code is beyond the primary scope of this work, which aimed to establish the feasibility of fully automated development.

Q6: Hardware abstraction, interface diversity, and framework generality beyond tested tasks. A6: AutoEmbed achieves generality without a fixed HAL through: 1) Hardware configuration (Sec 3.1.1) capturing platform details. 2) Automated library solving identifying suitable libraries for diverse hardware. 3) Knowledge generation learning library usage by parsing files, adapting to varied interfaces. Our evaluation on 71 modules, 4 MCUs, and 355 tasks shows this adaptability.

Q7: Explicit discussion of limitations and threats to validity. A7: We acknowledge the suggestion for an explicit 'Limitations' section. While currently implicit (e.g., future work in Sec. 6 hinting at current scope, challenges in Sec 1.2 framing the tackled problem), we will add a dedicated limitations discussion if accepted.

Reviewer #50B

Q1: How to check logical errors. A1: We use a validator to verify the generated code's logic. During code generation, DEBUG statements are added after each functional step. On execution, the validator analyzes the DEBUG output to catch logical errors missed by compilation, ensuring correct system performance.

Q2: How metrics reflect the effectiveness of individual components. A2: Coding accuracy (correct API usage) and completion rate (task success in one try) are end-to-end metrics. Their improvement in ablation studies (Fig. 13a) demonstrates individual component effectiveness: better library resolution ("+LS") and knowledge generation ("+KG") lead to higher accuracy and completion rates.

Q3: Elaborate on classification of tasks. A3: EmbedTask tasks define **functional requirements for IoT applications** (e.g., "Read sensor, display data". Difficulty levels (1-3) are based on **Task Complexity = Number of Functionalities (N_f) × Number of Components (N_c)**, with complexity increasing from Level 1 to Level 3, as shown in Fig. 11. Task examples are provided in the 'Example_tasks.json' file with the submitted code.

Q4: Clarify intended users. A4: AutoEmbed serves both beginners, by lowering the entry barrier, and professionals, by reducing repetitive manual work and shortening development cycles. This versatility makes it suitable for diverse applications, including hobbyist projects, education, or rapid prototyping in industry.

Reviewer #50C

Q1: Explanation for error bars in Figure 13. A1: Figure 13 error bars show the standard deviation from 10 experimental repetitions per data point, as per Sec 4.1 (Metrics). This indicates performance consistency. We will clarify this in the revision.

Q2: Rationale for Knowledge Generation (Sec 3.2) over LLM fine-tuning. A2: We chose dynamic knowledge extraction over LLM fine-tuning due to the embedded domain's vast and rapidly evolving libraries (over 7,000 noted in Sec 1.2). Our approach (Sec 3.2) adapts to new or updated libraries by extracting precise API details directly from their files without continuous, costly fine-tuning. This ensures up-to-date knowledge crucial for hardware interaction. Furthermore, our selective memory pick-up leverages this efficiently, reducing tokens and cost (Table 3).

Q3: Cost reduction calculation details. A3: The stated cost reductions are based on measured token consumption (Table 3) and the LLM provider's pricing at the time of our experiments.

Q4: Motivation for case studies. A4: The case studies demonstrate AutoEmbed's capability on intricate, multi-component systems relevant to real-world applications and evaluate performance with different LLMs on these complex tasks.

Q5: Is the gap left by related work Ref. [21] explicitly stated to motivate AutoEmbed? A5: Yes, Ref. [21] is the most direct prior work on LLMs for embedded systems, as discussed in Sec 1.1 and Sec 5. We note it primarily explores human-in-the-loop assistance and lacks full automation. AutoEmbed addresses this gap by being the first fully automated system for this domain, which is a core part of our motivation.