Embedded Technology and Intelligent Systems 嵌入式技术与智能系统, 2025, 2(6), 337-347
Published Online December 2025 in Hans. https://www.hanspub.org/journal/etis
https://doi.org/10.12677/etis.2025.26033

Hans汉斯

大模型在嵌入式软件开发中的应用

董岩博，蒋立伟，林金龙

北京大学软件与微电子学院，北京

收稿日期：2026年2月2日；录用日期：2026年5月1日；发布日期：2026年5月27日

摘　要

嵌入式软件与硬件和领域知识高度相关，开发难度大，开发周期长。以Transformer架构为核心的大语言模型(LLMs)凭借强大的代码理解与生成能力，为提升嵌入式软件开发效率提供了可能。文章介绍大语言模型在嵌入式软件开发中的应用现状，包括应用过程、方法和常用的大语言模型及其技术特性，并重点梳理大模型在嵌入式软件开发中的应用场景，以及嵌入式软件代码缺陷检测的核心技术路径、场景适配方案。文章通过综述现有研究成果，为嵌入式软件研究和开发人员应用大模型提供参考，助力嵌入式软件开发方法的智能化转型。

关键词

大语言模型，嵌入式软件，代码生成，代码缺陷检测

Applications of Large Language Models in Embedded Software Development

Yanbo Dong, Liwei Jiang, Jinlong Lin

School of Software & Microelectronics, Peking University, Beijing

Received: February 2, 2026; accepted: May 1, 2026; published: May 27, 2026

Abstract

Embedded software is highly correlated with hardware and domain knowledge, featuring high development difficulty and long development cycles. Large Language Models (LLMs) centered on the Transformer architecture, with their robust capabilities in code understanding and generation, have made it possible to improve the efficiency of embedded software development. This paper introduces the current application status of large language models in embedded software development, including the application processes, methods, commonly used large language models and their technical characteristics. It also focuses on sorting out the application scenarios of large models in embedded

文章引用: 董岩博, 蒋立伟, 林金龙. 大模型在嵌入式软件开发中的应用[J]. 嵌入式技术与智能系统, 2025, 2(6): 337-347. DOI: 10.12677/etis.2025.26033

software development, as well as the core technical approaches and scenario adaptation schemes for code defect detection in embedded software. Through a review of existing research findings, this paper provides a reference for embedded software researchers and developers in the application of large models, and facilitates the intelligent transformation of embedded software development methodologies.

Keywords

LLMs, Embedded Software, Code Generation, Code Defect Detection

1. 引言

嵌入式系统已广泛渗透于工业控制、智能终端、汽车电子、物联网等关键领域，其软件开发质量与效率决定着终端产品的性能与竞争力。嵌入式软件开发需要同时兼顾硬件资源约束、实时性要求、底层驱动适配与上层应用逻辑，导致开发周期长、调试难度大，对开发者的综合能力要求较高。随着物联网技术的普及，嵌入式设备数量激增且场景愈发多元，传统依赖人工经验的开发模式已难以应对规模化、定制化的开发需求，亟需新兴技术赋能产业升级。

以 Transformer 架构为核心的大语言模型(LLMs)凭借千亿级参数规模与海量数据预训练优势，可以实现对自然语言与编程语言的深度理解[1]。从通用代码生成、智能调试到需求文档自动化生成，GitHub Copilot 等工具的普及使编码效率大大提升，也验证了大语言模型在代码编写及其他软件开发任务中的巨大潜力。

当前，大语言模型在嵌入式软件开发中的应用仍处于探索阶段，对于这一技术的研究已经取得一定阶段性成果，本文将现有的大语言模型应用于嵌入式开发的研究成果进行整合，对现有的研究方法进行介绍，并对未来的发展方向进行展望，本文将有助于相关领域研究者系统探究大语言模型与嵌入式软件开发的融合路径，明确其适用场景与技术边界，构建高效的人机协同开发框架，为推动嵌入式开发模式智能化转型提供技术参考，助力嵌入式产业在智能化浪潮中实现高质量发展。

2. 代码大语言模型

代码大语言模型(Code LLMs)和通用大语言模型一样，采用 Transformer 架构，其预训练的核心数据是大规模无标签代码语料库，仅搭配少量文本和数学数据，而通用大语言模型的预训练则以大规模文本数据为主，仅融入少量代码和数学数据以提升逻辑推理能力。部分代码大语言模型(如 Qwen2.5-Coder [2])会在训练过程中引入合成数据。

与通用大语言模型类似，代码大语言模型也可分为三类架构：仅编码器型、仅解码器型、编码器-解码器型。仅编码器型通常用于生成代码理解类任务，例如类型预测、代码检索、克隆检测等；仅解码器模型主要用于生成类任务中，例如代码生成、翻译、摘要等；编码器－解码器模型可完成代码理解和生成两类任务，但在特定任务上表现未必优于前两者[1]。

代码大语言模型中 Transformer 的关键模块包括：多头自注意力模块——捕捉语义关系、位置感知前馈网络——优化序列嵌入位置、残差连接与归一化——缓解梯度问题、位置编码——补充位置信息[1]。

代码生成专用大语言模型(LLMs for code generation)指利用大语言模型从自然语言描述生成源代码的技术，这一过程也被称为“自然语言到代码(NL2Code)”任务。输入是自然语言描述，包括编程问题陈述，也可包含编程上下文(如函数签名)，而后模型生成对应源代码，生成的代码再通过编译器/解释器执行单元测试，根据执行反馈(如错误信息)迭代优化先前生成的代码，形成闭环优化流程。

代码生成任务所采用的 Transformer 架构主要分为两类：编码器–解码器架构和仅解码器架构。编码器–解码器架构同时包含编码器和解码器，编码器负责处理输入数据并生成表征，解码器基于这些表征生成输出；仅解码器架构仅保留 Transformer 的解码器部分，通过单一层堆叠同时完成输入处理和输出生成。因此，编码器–解码器架构适用于需要在不同输入和输出域之间建立映射的任务，而仅解码器架构则专门针对序列生成和续写类任务设计。

3. 大模型嵌入式软件开发模式

3.1. 可行性探究

对于大模型在嵌入式开发中应用的可行性探究工作，涉及开发全流程中的诸多领域。大模型具有极强的代码生成理解能力，因此较早实现验证的便是大模型辅助代码生成的可行性，并且为了完成对大模型生成的嵌入式代码的功能评估，诸多的测试和评估方法也应运而生；一些研究团队通过对大模型在嵌入式系统中的能力与局限的探究，发现大模型的跨软硬件的推理能力也是其核心优势之一，其在嵌入式开发中的价值不仅局限于代码生成，这一优势也使得大模型在更多开发环节中的应用部署具备可能，为全开发流程自动化提供可能[3]。资深固件工程师 Mark 分享的 AI 辅助嵌入式开发实操指南，通过精准配置 AI 工具，全程聚焦 STM32、ESP32 等主流硬件的实操需求，将大模型适配嵌入式场景，覆盖到代码生成、调试、优化、测试全流程，能够解决嵌入式开发中重复编写外设初始化代码、寄存器配置查询繁琐、硬件 bug(如 I2C 时序、SPI 间歇性故障)排查耗时、新手频繁出现同质化问题的痛点，并且可以将嵌入式系统程序调试时间减少 60%，同时提升代码可靠性，完美验证了 LLMs 辅助嵌入式开发的可行性和高效性[4]。Englhardt 等人通过构建首个开源可扩展的硬件在环评估框架，将 LLM 生成的代码直接上传到微控制器(如 Arduino Uno、nRF52832)，通过“传感器–执行器对”与人类编写的基准代码进行物理输出对比，验证功能正确性。基于该框架完成 450 次真实硬件实验，覆盖光敏电阻、超声波测距仪、6 轴 IMU 等典型嵌入式场景，针对 GPT-3.5、GPT-4、PaLM 2 三个主流模型，从硬件理解、代码生成、系统设计三个维度展开深度测试，发现跨域知识与代码生成能力 GPT-4 表现最优，在部分任务中单次提示即可生成完全正确的代码，在 50 次实验中 66%能生成可用的 I2C 接口，还能编写寄存器级驱动、LoRa 通信代码；针对 nRF52832 的电源优化建议，使电流消耗降低 740 倍至 12.2 μA (接近人类专家优化的 8.6 μA)。硬件规格理解方面，GPT-4 能精准识别常见芯片(如 BME280、LSM6DSO)的 I2C 地址、寄存器功能、通信协议，甚至能解释 datasheet 中的位字段含义；对虚构传感器的寄存器表，可生成符合规格的驱动伪代码。并且即使代码不完全正确，LLM 仍能提供 actionable 建议，如检查接线、修正 I2C 地址、优化采样率等，尤其对硬件新手帮助显著[3]。

3.2. 框架概述

当前大模型在嵌入式软件开发中的应用主要围绕在代码生成与调试阶段，通过构建实现“需求描述–生成代码–编译验证–问题优化”的闭环框架来完成开发任务。Englhardt 等人提出的人机协同开发模式[3]，将大模型作为辅助工具，通过结构化工作流整合人类领域知识与大模型的代码生成能力，覆盖需求分析、代码生成、调试部署全流程：(1) 首先要求用户提供完整上下文，如硬件型号、引脚连接、库版本、预期行为，避免模糊表述；(2) 将编译器错误信息反馈给 LLM；(3) 详细描述物理行为异常，利用

LLM 可关联硬件逻辑提供解决方案的能力；此外，还可以考虑拆分复杂任务，在模块化生成函数后再进行人工整合，还可以提供示例代码约束输出、及时更新代码上下文，避免 LLM 陷入数值计算错误。该团队通过使用该方法，使得任务成功率显著提升，LoRa 环境传感器的开发成功率从 25%提升至 100%，零硬件/C++经验的用户可在 40 分钟内完成功能完整的收发器。新手借助 LLM 填补硬件知识缺口，专家通过 LLM 生成模板代码提升效率。经 NASA TLX 评分得到，参与者的任务感知负载从 4.67 降至 3.25，尤其减少了“硬件接线排查”“寄存器配置”等繁琐工作的负担[3]。

3.3. 跨平台嵌入式开发

跨平台嵌入式开发也是当前研究方向之一，应用大模型将一个平台的代码转化成适配另一个平台的代码并实现相同功能，用于节省跨平台开发需要从零做起的高时间成本。Xu 等人提出的方法通过外部知识检索补充大模型的硬件知识，利用编译器反馈修正语法错误，提升代码的跨平台适配性与正确性[5]。首先利用语言大模型检索增强生成，提取同类硬件的预生成代码作为参考，辅助大模型复用成熟 API 与逻辑，避免重复错误；再利用推理大模型实现编译器反馈优化，捕获框架的语法错误信息，反馈给大模型进行针对性修正，降低跨平台迁移的语法适配成本；最终基于硬件引脚映射、编程语言差异(如 Arduino →MicroPython)自动调整代码和电路连接方案。

4. 大模型代码生成

代码生成是大模型辅助嵌入式开发的核心场景，开发者可以使用精准的自然语言指令部署大模型使其实现嵌入式代码的生成。对于初步生成的代码所存在的部分语法、编译问题和功能缺失，还需要实现反馈优化，通过向大模型反馈其输出结果在运行中存在的问题，使其重新思考理解并生成优化代码，增强其生成能力，完成设计任务[3] [5]。并且不同 AI 工具有着不同的嵌入式专属配置，包括提示词设计、上下文配置、嵌入式训练方法(包括运行软件配置、禁用文件提示等)、大模型思考模式等等，不同的模型配置、设计框架、优化方法也会使得生成结果在实际应用测试中表现出较大差异。下面介绍面向不同应用场景的不同设计和优化方法，以及所使用的大语言模型所达到的生成效果。

4.1. 提示词设计

首先，通过系统提示词定义 LLM 的目标角色、核心能力和行为准则；然后，通过用户提示词明确 LLM 的具体任务——根据给定规格完成函数定义。并且在提示词中可以提供 C 程序的结构框架，包括头文件、全局变量和函数头等信息。

4.1.1. 语言规格

针对具体任务描述，从语言规格角度可以分为高层自然语言规格(HLNL)、低层自然语言规格(LLNL)和 ACSL 规格[6]。HLNL 聚焦系统需实现的核心功能与约束，明确系统要达成什么目标，如“若存在刹车灯请求，则应激活卡车车灯”；LLNL 目标是实现高层自然语言规格，明确系统如何实现目标的具体要求，如“若运行状态为正常运行或有限功能紧急停机，且供电电压未处于低电平，则应启用刹车灯”；ACSL 规格是以 ANSI/ISOC 规格说明语言(ACSL)编写的形式化规格说明集。

Patil M S 等人提出的 Spec2code 框架利用含前置条件、后置条件、帧条件的 ACSL 形式化规范与高、低层级自然语言规范来覆盖功能与约束定义，生成初始代码，再通过将编译器、验证工具等“批评者”提供的反馈用于提示词迭代，进行迭代优化与模型微调，持续优化代码。在使用 GPT-3.5 与 GPT-4-turbo，基于零样本链式思考，完成对三款真实汽车控制模块(转向油位检测、刹车灯激活、动力转向备份)的设计实现后，Patil 等人验证了 Spec2code 框架极简版本的可行性，证明无需复杂迭代也能生成可编译、可验

证的工业级代码也说明了技术细节明确的描述规格能提高所生成代码的可用性[6]。

4.1.2. 连接约束

除了给定任务描述外，还可考虑为大模型提供电路图，规定连接要求。也可以要求大模型实现跨平台设计，即要求大模型重新设计代码，将某一硬件平台的代码和原理图迁移至另一平台，实现相同功能。EmbedAgent 是一个嵌入式代码生成和评估方法[5]，其 EmbedBench 基准是三个核心任务场景。三个场景分别是：(1) 给定任务描述和电路原理图，生成嵌入式代码；(2) 仅给定任务描述，自主设计电路原理图并编写对应代码；(3) 给定某硬件平台代码和原理图，要求迁移至另一平台，设计实现代码。

设计结构化电路描述格式(.json)，将硬件连接转化为 LLM 可理解的“组件 ID + 引脚”配对形式，基于 Wokwi 虚拟电路仿真平台，实时监控虚拟硬件的运行状态(而非依赖串行输出)，直接验证生成代码是否符合任务要求[5]。

EmbedBench 基准包含 126 个手动构建的测试案例，覆盖 9 类电子元件，包括 LED、RGB LED、按键、7 段数码管、滑动变阻器等。每个案例含任务描述、参考解决方案和自动化验证用例，确保评估的全面性。Ruiyang Xu 等人在对 10 个主流 LLM 进行系统性测试后发现在给定原理图时，最优模型 DeepSeek-R1 的 pass@1 仅 55.6%；不指定原理图时降至 50.0%；跨平台迁移任务难度最高，ESP32 平台的 ESP-IDF 框架迁移最优 pass@1 仅 29.4% (Claude 3.7 Sonnet)，而 Raspberry Pi Pico 的 MicroPython 迁移表现相对较好(最优 73.8%)。DeepSeek-R1、Claude 3.7 Sonnet 等推理型 LLM 整体优于聊天型 LLM，但存在像为 7 段数码管生成二进制码时陷入冗余计算等“过度思考”的问题；DeepSeek-V3、Llama-3.3-Instruct 等聊天型 LLM 难以灵活适配任务场景，如无法将预训练的 7 段数码管编码表调整为共阳极配置；DeepSeek-R1-Distill 等 SFT 蒸馏型 LLM 稳定性差，面对陌生任务时性能甚至低于基础模型[5]。QWQ 等推理型 LLM 在自主设计电路时，会选择比参考方案更合理的引脚连接方式，导致其在不提供电路图时的性能更优，说明僵化的预设原理图可能限制 LLM 的推理能力。

4.1.3. 其他优化方法

针对无线通信的功耗、调制方案等专项需求，Medaranga 等人提出通过精准提示引导大模型生成反向散射发射器等无线模块代码，确保通信协议合规性与信号稳定性的方法[7]。主要技术包含：(1) 专项参数明确：即在提示中指定调制方案、中频、比特率、硬件平台等关键参数，避免大模型幻觉；(2) 硬件特性适配：结合反向散射通信的低功耗原理，生成 GPIO 引脚精准控制逻辑，确保信号调制和传输符合无线标准。

针对传统手动方法的低效问题和现有 LLM 在领域适配性上的不足，Lin 等人提出一种“监测－分析－规划－执行－知识”(MAPE-K)架构，设计面向航空嵌入式系统的 LLM 结构化流程[8]。通过“分类－形式化－组装”三步法 + BNF 语法约束，解决 LLM 的领域适配性问题。以真实子系统为案例，对比不同参数 LLM、手动方法、直接 LLM 方法，量化证明了所提方法在效率和准确性上的优势，具备实际工程应用价值。主要技术包含(1) 领域知识建模：将航空系统需求划分为监测、决策、通信等 5 大领域，定义 11 类核心操作，明确其语法规则与硬件交互逻辑；(2) 需求规范化：采用巴科斯－诺尔范式(BNF)统一需求表达，包含时间约束、知识引用等关键元素，消除自然语言歧义；(3) 设备信息提取、功能规范形式化(分类、形式化、组装)、约束提取，以确保代码与硬件交互、任务逻辑一致[8]。

此外，基于零样本思维链提示技术设计提示词，可以引导模型在生成最终答案之前，先逐步推导出一系列的中间推理步骤，从而提升大语言模型在复杂推理任务上的表现[6]。

在编译器反馈阶段，针对平台的语法错误问题，将编译器的错误信息反馈给 LLM，弥补大模型在硬件知识、跨平台适配等方面的不足，通过人类参与、工具反馈、知识检索等方式引导其进行修正，实现

优化。EmbedAgent 基于这种方法仅需 1 轮迭代，便可实现语法错误率从 34.1%降至 3.2%，ESP32 迁移任务的 pass@1 从 21.4%提升至 27.8% [5]。

4.2. 生成代码评估方法

对于编译成功的代码，可以从以下角度进行评估：功能正确性、程序等价性和代码质量。功能正确性旨在验证程序的正确性，即是否满足规格说明；程序等价性则是利用已有的代码库中人工编写的代码和所生成的代码进行对比，捕捉二者的语义差异，检验其相对正确性；代码质量可以通过代码行数或其他特定的规则如函数规模、循环要求等进行考量。Patil 等人在 Spec2code 框架中便是使用 Frama-C 中的最弱前置条件演算插件(WP)，基于 ACSL 规格验证程序的功能正确性，采用 Z3 和 Alt-Ergo 作为后端求解器；通过 diffkemp 工具验证功能正确性，该工具可对 LLVM 字节码形式的程序进行指令级等价性验证，验证过程采用一组语义保留转换技术；采用定量与定性相结合的方式评估代码质量，除统计对比代码行数外，还人工检查其对“十大规则”的符合程度[6]。

4.3. 全自动化平台构建

对于整个开发工作，目前有研究致力于实现构建端到端自动化嵌入式开发全流程。其中 EmbedGenius 就是一个面向通用嵌入式 IoT 系统的全自动化开发框架，包含端到端自动化嵌入式开发全流程，解决硬件依赖解析、库知识缺失、编程复杂性三大痛点，无需人工干预即可完成从需求到部署的闭环[9]。核心技术包括(1) 硬件依赖解析：基于名称匹配、版本迭代频率、架构兼容性评分，自动选择最优硬件库，适配多模块组合场景；(2) 库知识注入：从库的头文件与示例中提取 API 声明、参数、使用顺序，构建 API 表与组件功能表，注入大模型内存；(3) 嵌套反馈循环：通过“编译循环”(确保语法正确)与“闪存循环”(验证功能正确性)的嵌套反馈，结合 DEBUG 信息实现错误自修正；(4) 安全防护：对高风险操作进行开发者确认，对敏感信息进行占位符替换。Yang 等人通过“大规模基准测试 + 实际案例验证”，全面验证 EmbedGenius 的性能，包括 4 个主流开发平台(Uno R3、NUCLEO-L4、Nano RP2040、Nano ESP32)、测试四种大模型性能(GPT-4o、GPT-4、Claude-3.5、GPT-3.5)、使用 71 个硬件模块(传感器、通信模块、显示器等)、355 个 IoT 任务(分 3 个难度等级，覆盖环境监测、运动控制、数据通信等场景)。以 3 个需人类参与的开发方法(LLM-Prompt、Duinocode、LLM-direct)作为对比，从编码准确率(API 调用与参数配置的正确性)、任务成功率(一次执行完成所有子功能的概率)两个角度评估其性能。最终得出 GPT-4o 综合性能、延迟、成本最优，支撑系统达到 95.7%编码准确率、86.5%任务成功率的核心指标；GPT-4/Claude-3.5 在更强的长文本解析(如复杂库文件)或稳定性需求下可替换使用，但需接受更高延迟/成本；而 GPT-3.5 则需避免用于复杂场景：仅推荐用于“单传感器读取”“LED 控制”等简单任务，或对成本极度敏感的批量开发场景。论文还通过两个真实场景案例，验证系统的实用性：环境监测系统：整合 DHT11 传感器、LoRa 模块、OLED 显示器，2.6 分钟完成开发，部署于森林持续监测温度，支持多 LLM (GPT-4o、Claude-3.5 等)，成功率 ≥80%；远程控制系统：整合 PIR 人体传感器、继电器、Wi-Fi 模块，3.1 分钟完成开发，部署于智能家居实现风扇自动控制，自动生成实时更新的 Web 控制界面，编码准确率 ≥90% [9]。

EmbedAgent 也构建出一个端到端自动化流水线，通过开发自动化提交机器人，将 LLM 生成的原理图和代码映射到 Wokwi 的虚拟环境中，自动执行测试用例并返回结果，无需手动组装硬件，便可实现高效、可扩展的评估[5]。

5. 大模型生成的嵌入式代码的检测方法

嵌入式系统广泛应用于工业控制、汽车电子等关键领域，代码安全性直接决定设备可靠性，嵌入式

系统中约 73%的安全漏洞源于代码缺陷[10]，嵌入式领域安全漏洞中 32%源于缓冲区溢出，25%涉及内存泄漏[10]。根据 NIST SP 800-53 的安全控制框架，这些漏洞类型对应访问控制、边界保护等安全控制族[11]。嵌入式系统的硬件相关性(如寄存器操作、中断机制)与资源约束(内存、算力有限)，使得大模型生成的代码易存在隐性缺陷与显性错误。

5.1. 嵌入式代码缺陷检测

嵌入式代码缺陷特指因硬件交互逻辑不当、资源分配不合理或规格实现偏差导致的潜在问题，如中断竞态条件、寄存器配置错误、缓冲区溢出等，其检测需深度关联嵌入式领域特性与代码语义。本节分析数据集增强与 RAG 检索增强两类检测技术的进展。

5.1.1. 数据集增强与领域适配路径

通用代码数据集难以覆盖嵌入式系统的硬件交互特性，如 MCU 寄存器操作、汇编/C 混合编程，数据集的领域适配与规模扩充成为提升缺陷检测精度的基础。Bhandari 等人提出的 IoTvulCode 框架[12]是该路径的代表性研究，其核心思路是构建面向 IoT 嵌入式系统的大规模标注数据集，通过 FlawFinder、CppCheck、Rats 三款静态分析工具联合扫描 Linux-rpi、FreeRTOS 等 11 类主流 IoT 项目的源代码，最终形成包含 1,014,548 条语句(65,135 条漏洞样本)和 548,089 个函数的高质量数据集，标注遵循 CWE 标准，支持二进制分类(漏洞/无漏洞)与多分类(具体漏洞类型)任务。在模型设计上，该框架采用 RNN、LSTM 等序列模型，针对嵌入式代码短语句(平均 38 字符)的特性优化输入长度与词嵌入策略，最终实现语句级二分类准确率 99.1%、召回率 88.5%，多分类任务中准确率、精确率与召回率均达 99%，显著优于 iDetect 等依赖小数据集的传统工具，其数据集规模较 iDetect 扩大 162.4 倍，有效缓解了嵌入式领域标注数据稀缺的问题。

该路径的创新延伸集中在数据集场景化优化与模型轻量化适配。Tang 等人在 ICFEM 2023 的研究[13]基于 IoTvulCode 的数据集构建思路，筛选 ESP32、Raspberry Pi 等物联网硬件的专属代码样本，对 CodeLlama-7B 进行领域微调，通过归一化不同厂商 MCU 的宏定义与寄存器命名，减少代码变体对模型的干扰，最终在 IoT-Security-DataSet 上实现漏洞检测 F1 值 81.7%，较通用 CodeLlama 提升 25%。此外，Hanif 等人的 VulBERTa 模型[14]针对嵌入式 C/C++代码的语法特性优化分词策略，保留寄存器命名、宏定义的完整性，在 IoTvulCode 数据集上的缺陷检测 F1 值达 89.2%，较通用 BERT 模型提升 17%，验证了数据集与模型结构协同优化的有效性。

5.1.2. RAG 增强的多阶段静态检测路径

嵌入式代码缺陷常隐藏于规格说明与代码实现的语义偏差中，传统静态分析难以关联非结构化规格文档(如硬件手册、协议规范)，RAG (Retrieval Augmented Generation)技术通过检索增强实现规格与代码的语义对齐，成为解决该问题的核心路径。Qayyum 等人在 ACM TDES 上提出的 3 阶段 4 步骤方法[15]是该路径的典型代表，其技术核心在于将 RAG 与迭代信息补充相结合：首先通过预处理将规格文档拆分为 500 token 的片段并嵌入向量库，再对目标 Verilog 代码逐行进行语义检索，获取相关规格片段作为 LLM 上下文；若初始检测失败，则进入 4 阶段 bug closure 流程——逐步增大规格文档检索块至 1000 token、补充前后各 5 行相邻代码、将复杂的硬件逻辑表达式扁平化拆解为原子级子表达式、提取功能属性，通过多维度信息补充缩小缺陷定位范围。该方法针对 AES、I2C、USB 等 5 类 OpenTitan IP 的 9 类缺陷(含常量错误、切片错误、敏感列表错误等)进行测试，最终缺陷修复率达 60%，另有 11%的缺陷被准确识别但未完成修复，成功解决了传统 RAG 对常量类缺陷检测失效的问题[16]。

类似地，Qayyum 等人在 LAD 2024 的研究[16]进一步简化该路径，通过 RAG 检索规格文档与代码

上下文，针对 AON 定时器、UART、EDN 三类 OpenTitan IP 的功能缺陷，实现除常量错误外 100%的修复率，验证了该路径在专用嵌入式 IP 检测中的适配性。该路径的创新延伸主要体现在检索维度的拓展与硬件知识的深度融合：Englhardt 等人在 arXiv 2023 的研究[3]通过 RAG 补充 MCU 硬件手册、寄存器配置表等领域知识，评估 GPT-4、CodeLlama 等模型对嵌入式特有缺陷的检测能力，发现增强后模型对中断处理漏洞、寄存器配置错误的检测准确率从 78.3%提升至 89.1%，硬件交互类缺陷漏检率从 42%降至 28%。Tang 等人的研究[13]则将 RAG 检索范围扩展至 IoT 设备的通信协议文档(如 SPI、I2C 协议规范)，使模型能精准识别协议交互中的逻辑缺陷，在工业物联网网关代码检测中，协议相关缺陷的识别准确率较通用 RAG 提升 32%，进一步拓宽了该路径的应用场景。

5.2. 嵌入式代码 bug 检测

嵌入式代码 bug 特指导致系统运行异常的显性错误，如语法错误、内存访问错误、编译错误等，其检测需兼顾实时性与修复精准性，同时适配嵌入式系统的资源约束。本节讨论自反思迭代修复与数据集优化方法的技术进展与工程实践效果。

5.2.1. 自反思驱动的迭代修复路径

大模型生成的嵌入式代码 bug 常伴随编译器反馈缺失或上下文信息不足的问题。自反思机制通过引入编译器错误信息、迭代优化修复方案，成为提升 bug 检测与修复效率的关键路径。Cui 等人提出的 OriGen 框架[17]是该路径的标杆之作，其技术创新体现在两个核心层面：一是代码到代码的数据增强策略，通过 Claude3-Haiku 对开源 RTL 代码进行功能描述生成与重构，过滤超过 300 行或非标准格式的低质量样本，同时收集编译失败案例构建包含“错误代码–编译器反馈–修复代码”的错误修复数据集；二是双 LoRA 模型架构与自反思循环，Gen LoRA 负责基于自然语言指令生成初始 RTL 代码，Fix LoRA 在编译器反馈触发下，分析语法错误、模块接口错误等信息并迭代修复代码，直至通过 Icarus Verilog 编译器验证。该方法在 VerilogEval-Human 基准上 pass@1 达 54.4%，超越 GPT-4 Turbo，在 VerilogFixEval 基准(编译错误修复专项)中，自反思修复能力较 GPT-4 提升 19.9%，成功解决了开源 LLM 在 RTL 代码语法错误、寄存器传参错误修复上的短板。

该路径的创新延伸聚焦于反馈机制优化与端侧部署适配。Tsai 等人提出的 RTLFixer 框架[18]将自反思与 ReAct 策略结合，通过 RAG 检索人类专家修复案例库，为 LLM 提供针对性修复指导，针对 RTL 语法错误的修复成功率达 83%，较无指导自反思提升 27%。Thakur 等人的 AutoChip 框架[19]则直接将编译器错误信息输入 LLM，无需检索外部数据库，通过多轮对话引导模型定位 bug 根源，在 8 位累加器微处理器设计中，编译错误修复效率较传统方法提升 4 倍，且修复代码的硬件资源占用率降低 12%。针对嵌入式端侧部署需求，OriGen 通过模型量化与剪枝技术，将 Fix LoRA 模块的推理耗时控制在 200 ms/千行代码以内，可直接部署于 ARMCortex-M4/M7 系列 MCU，为端侧实时 bug 检测与修复提供了可行方案。

5.2.2. 数据集优化的批量 bug 检测路径

嵌入式代码 bug 的多样性，如 MCU 架构差异、编程语言特性等要求检测模型具备强泛化能力，而高质量大规模数据集是实现这一目标的核心支撑。Bhandari 等人的 IoTvulCode 框架[12]在 bug 检测领域的拓展应用，充分体现了数据集优化的价值。该框架构建的数据集覆盖 11 类主流 IoT 项目，包含 65,135 条 bug 样本(涵盖缓冲区溢出、整数溢出、空指针等 10 类主要 CWE 类型)，通过 srcML 工具提取函数级与语句级代码片段，结合 Guesslang 进行编程语言分类，为批量 bug 检测提供了标准化数据支撑。在模型层面，采用 RNN、LSTM 等序列模型，针对嵌入式代码短语句、高复用性的特点优化输入长度(语句级 150 token、函数级 1024 token)与词嵌入策略，最终实现语句级 bug 检测二分类准确率 99.1%、召回率 88.5%，多分类准确

率 99%，较 iDetect 等传统工具的损失值降低 78%，展现出大规模数据集对模型泛化能力的提升作用。

该路径的创新主要体现在数据集场景化细化与模型适配优化。Englhardt 等人在 arXiv2023 的研究[3]通过数据集扩充，加入 STM32、Arduino 等平台的运行时 bug 样本(如定时器中断冲突、SPI 通信时序错误)，使 GPT-4 对这类嵌入式特有 bug 的检测准确率从 62%提升至 79%，验证了数据集场景化优化的重要性。此外，针对嵌入式系统的实时性要求，部分研究通过数据集筛选，保留短代码片段(≤50 行)的 bug 样本，训练轻量化模型，使检测耗时控制在 10 ms/行以内，满足嵌入式开发的实时检测需求。

5.3. 大模型检测技术的场景适配性分析

不同嵌入式应用领域的硬件特性、功能需求与安全标准存在显著差异，对大模型检测技术的适配性提出了差异化要求。为直观呈现各场景下的最优检测策略及性能表现，我们用下表从应用领域、核心约束、典型漏洞类型、推荐检测方案及代表成果与性能五个维度，系统梳理大模型检测技术的场景适配情况。

Table 1. Scenario adaptability analysis of large model detection technology
表 1. 大模型检测技术的场景适配性分析

应用领域	核心约束	典型漏洞类型	推荐检测方案	代表成果与性能
工业控制	高可靠性、实时检测、多架构适配	硬件交互类、内存溢出	EmbedVulDet 混合模型 + 边缘计算节点	F1 值 0.94，误报率 < 5%，检测耗时 < 200 ms/段
汽车电子	功能安全合规(ISO 26262)、漏洞可复现	实时性冲突、CAN 总线协议漏洞	TraceBERT + 动态轨迹分析	漏洞根源定位准确率 89%，支持 ISO 26262 报告生成
消费电子(MCU)	低算力、低功耗、轻量化部署	控制流异常、简单内存漏洞	量化压缩 CodeBERT (4-bit) + 静态分析	模型体积 < 100 MB，准确率 87.5%，支持本地检测
航空航天	极高安全性、全链路溯源、零误判	复杂逻辑漏洞、冗余代码引发的性能问题	多模态大模型 + 人工复核闭环	准确率 99.2%，支持漏洞关联影响分析

5.4. 现存核心问题与技术挑战

5.4.1. 技术层面的核心瓶颈

1) 开源数据集质量与规模不足：嵌入式领域高质量标注数据稀缺，IoTvulCode 数据集虽达百万级样本，但小众硬件相关样本占比不足 5% [12]。

2) 跨架构适配能力有限：现有模型在 ARM 与 RISC-V 架构间迁移检测准确率下降 25%~30%，难以适配异构硬件[12]。

3) 编译器反馈利用率低：开源模型对编译错误信息的解析能力薄弱，OriGen 的 self-reflection 机制虽有改善，但复杂错误修复率仍低于 40% [14]。

4) 功能规格书依赖性强，模糊或缺失规格会导致准确率下降 15%~20% [3]。

5.4.2. 未来研究方向与突破路径

1) 硬件感知的专用大模型构建：通过“通用代码预训练 - 硬件知识微调 - 场景数据精调”三级框架，提升异构硬件适配能力[12]。

2) 小样本与零样本检测技术创新：基于 Prompt Tuning 等方法，通过漏洞模式模板 + 少量标注样本实现高效检测，50 条数据场景已验证可行性[13]。

3) 轻量化与边缘部署优化：采用剪枝、量化等技术，让模型体积减少，检测耗时降低，准确率提升

[12]。

4) 工程化工具链集成：开发与 Keil、IAR 等环境集成的插件，实现“编码－编译－检测－修复”无缝衔接[14]。

6.展望

大模型在嵌入式软件开发中的应用已经开始展现巨大潜力，在代码生成、驱动开发与缺陷检测等核心环节，有效缓解了硬件适配复杂、专业知识依赖度高的传统痛点。但当前技术仍存在不少关键问题，未来可关注三个核心的方向来深化探索。

在技术适配层面，应重点突破跨架构与硬件感知的短板。现有模型在 ARM 与 RISC-V 等异构硬件间迁移时性能下滑明显，对小众硬件的支持不足，后续可以通过“通用预训练－硬件知识微调－场景精调”模式，补充寄存器配置、通信协议等专属知识，结合小样本学习降低标注数据依赖。同时优化编译器反馈解析机制，提升模型对复杂编译错误的修正能力，减少开发中的语法与逻辑漏洞。

在场景与工具层面，推进垂直领域定制与全流程整合。汽车电子、工业控制等领域对合规性、实时性要求差别很大，需要做针对性优化：汽车电子重点关注 ISO 26262 合规性检测，工业控制主要提升硬件交互缺陷识别精度，消费电子则通过剪枝、量化技术优化轻量化部署。工具链方面，推动大模型与 Keil、IAR 等开发环境集成，结合虚拟仿真平台构建“编码－检测－修复”自动化流程，优化人机协同模式，让开发者聚焦核心逻辑，模型承担重复编码与基础调试。

安全保障层面，完善漏洞防控体系。嵌入式代码直接影响设备可靠性，需建立更全面的漏洞分类标准，提升对内存溢出、实时性冲突等核心问题的“检测－修复”效率，避免大模型生成代码引入隐蔽风险，确保系统运行稳定。

总体而言，大模型对嵌入式软件开发有提升效率、整合流程的巨大潜力，但跨架构适配性不足、工具链无法衔接等问题仍需解决。未来研究需紧扣实际开发流程中的问题，通过技术优化、工具整合与安全防护，推动大模型在嵌入式软件开发各环节的深度参与并发挥重大作用。

参考文献

[1] Jiang, J., Wang, F., Shen, J., *et al*. (2024) A Survey on Large Language Models for Code Generation. https://dl.acm.org/doi/full/10.1145/3747588

[2] Hui, B., Yang, J., Cui, Z., *et al*. (2024) Qwen2.5-Coder Technical Report. https://arxiv.org/abs/2409.12186

[3] Englhardt, Z., Li, R., Nissanka, D., Zhang, Z., Narayanswamy, G., Breda, J., *et al*. (2024) Exploring and Characterizing Large Language Models for Embedded System Development and Debugging. *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, Honolulu, 11-16 May 2024, 1-9. https://doi.org/10.1145/3613905.3650764

[4] Stop Wasting Hours Debugging: Use AI to Write Better Embedded Systems Code. https://markaicode.com/ai-embedded-systems-code/

[5] Xu, R., Cao, J., Wu, M., *et al*. (2025) Embed Agent: Benchmarking Large Language Models in Embedded System Development. https://arxiv.org/abs/2506.11003

[6] Patil, M.S., Ung, G. and Nyberg, M. (2024) Towards Specification-Driven LLM-Based Generation of Embedded Automotive Software. In*: Lecture Notes in Computer Science*, Springer, 125-144. https://doi.org/10.1007/978-3-031-75434-0_9

[7] Medaranga, P., Shah, D., Kandala, S.V. and Varshney, A. (2024) POSTER: Simplifying the Networking of Wireless Embedded Systems Using a Large Language Model. *Proceedings of the ACM SIGCOMM* 2024 *Conference*: *Posters and Demos*, Sydney, 4-8 August 2024, 78-80. https://doi.org/10.1145/3672202.3673752

[8] Lin, J., Luo, Y., Chen, X., Gu, B. and Jin, Z. (2025) Automatic Generation of Structured Requirements for Aerospace Embedded Systems Using LLMs. 2025 *IEEE* 33*rd International Requirements Engineering Conference Workshops* (*REW*), Valencia, 1-5 September 2025, 181-188. https://doi.org/10.1109/rew66121.2025.00028

[9] Yang, H., Li, M., Han, M., *et al*. (2024) Embed Genius: Towards Automated Software Development for Generic

Embedded IoT Systems. https://arxiv.org/abs/2412.09058

[10] MITRE Corporation and SANS Institute (2024) 2024 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

[11] National Institute of Standards and Technology (2022) Security and Privacy Controls for Information Systems and Organizations (NIST SP 800-53 Revision 5). https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final

[12] Bhandari, G.P., Assres, G., Gavric, N., Shalaginov, A. and Grønli, T. (2024) IoTvulCode: AI-Enabled Vulnerability Detection in Software Products Designed for IoT Applications. *International Journal of Information Security*, **23**, 2677-2690. https://doi.org/10.1007/s10207-024-00848-6

[13] Yang, Y.L. (2023) IoT Software Vulnerability Detection Techniques through Large Language Model. In: *Lecture Notes in Computer Science*, Springer, 285-290. https://doi.org/10.1007/978-981-99-7584-6_21

[14] Hanif, H. and Maffeis, S. (2022) VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. 2022 *International Joint Conference on Neural Networks* (*IJCNN*), Padua, 18-23 July 2022, 1-8. https://doi.org/10.1109/ijcnn55064.2022.9892280

[15] Qayyum, K., Jha, C.K., Ahmadi-Pour, S., Hassan, M. and Drechsler, R. (2025) LLM-Assisted Bug Identification and Correction for Verilog HDL. *ACM Transactions on Design Automation of Electronic Systems*, **30**, 1-28. https://doi.org/10.1145/3733237

[16] Qayyum, K., Hassan, M., Ahmadi-Pour, S., Jha, C.K. and Drechsler, R. (2024) From Bugs to Fixes: HDL Bug Identification and Patching Using LLMs and Rag. 2024 *IEEE LLM Aided Design Workshop* (*LAD*), San Jose, 28-29 June 2024, 1-5. https://doi.org/10.1109/lad62341.2024.10691874

[17] Cui, F., Yin, C., Zhou, K., Xiao, Y., Sun, G., Xu, Q., *et al*. (2024) OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection. *Proceedings of the* 43*rd IEEE/ACM International Conference on Computer-Aided Design*, New York, 27-31 October 2024, 1-9. https://doi.org/10.1145/3676536.3676830

[18] Tsai, Y.D., Liu, M. and Ren, H. (2024) RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Model. *Proceedings of the* 61*st ACM/IEEE Design Automation Conference*, San Francisco, 23-27 June 2024, 1-6. https://doi.org/10.1145/3649329.3657353

[19] Thakur, S., Blocklove, J., Pearce, H., *et al*. (2023) AutoChip: Automating HDL Generation Using LLM Feedback. https://arxiv.org/abs/2311.04887