IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, June 2026
Impact Factor: 8.2

Human-Driven vs. AI-Driven Development in Embedded and Operating Systems: Implications for Reliability, Security, and Maintainability

Supriya Mishra
Vidyavardhini's College of Engineering & Technology, Vasai Road

Abstract: *AI coding tools have become a routine part of software development, including in domains where mistakes can cause physical harm. This paper examines what happens when those tools are used to write software for Embedded Systems and Operating Systems — the code running inside medical devices, vehicle braking systems, power grids, and aircraft. We investigate three dimensions of risk: reliability (does AI code work correctly on real hardware?), security (does it introduce dangerous vulnerabilities?), and maintainability (can it be updated and fixed over its operational lifetime?). Drawing on four documented real-world incidents from 2025-2026 and survey data from over 200 embedded engineering professionals, we find that AI-generated system code exhibits a characteristic failure pattern we term Contextual Brittleness, reproduces historical memory-safety vulnerabilities at scale, and imposes a 66% overhead on long-term maintenance. Most critically, we document what we call the Closed Exploit Loop: AI generates embedded code containing memory-safety flaws, and AI-powered tools can now autonomously discover and exploit those same flaws without human involvement. We propose a four-layer Hybrid Governance Model that allows teams to use AI productively while preserving the safety standards that embedded engineering demands.*

Keywords: AI Code Generation, Compiler Correctness, CWE-787, Embedded Systems, Hybrid Governance Model, Linux Kernel Security, Memory Safety, Operating System Vulnerability, Software Reliability.

I. INTRODUCTION

Software that runs inside a car's braking system, a hospital heart monitor, or a factory's control systems is held to a completely different standard than an ordinary mobile app. This software — collectively called Embedded Systems and Operating Systems (OS) — must respond in real time, use minimal memory, and never fail. A bug in an embedded OS is not a user experience issue; it is a potential safety hazard.

Over the past three years, AI coding tools have gone from a curiosity to a standard part of development workflows. GitHub Copilot, Cursor, Amazon Q, and agentic systems like Claude Code are now used by 84% of developers according to a 2025 Stack Overflow survey of 49,000 professionals [15]. The embedded world has not been left behind: a RunSafe Security survey of over 200 engineers across the US, UK, and Germany found that 80.5% actively use AI tools at work, and 83.5% already have AI-written code running inside deployed production systems — including power grids, medical equipment, and industrial plants [4].

This creates a structural problem. AI writes code at machine speed, but the safety checks required by international standards — IEC 61508, ISO 26262, DO-178C — were designed for human-paced development. The validation processes have not scaled to match the generation speed. This paper investigates the consequences of that gap through four real incidents, industry survey data, and published benchmarks. It then proposes a practical Hybrid Governance Model for organisations that want the productivity benefits of AI without compromising the safety standards their embedded systems demand.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 11, June 2026 Impact Factor: 8.2

The rest of the paper is organised as follows. Section II reviews related work. Section III describes the research methodology. Section IV presents the four case studies. Section V discusses the results. Section VI proposes the governance model. Section VII concludes with recommendations and future work directions.

II. RELATED WORK

Research into AI-generated code quality spans three broad areas: productivity, correctness, and security. This section summarises the findings most directly relevant to system-level software.

A. Code Quality and Defect Density

CodeRabbit (2025) analysed 470 open-source GitHub pull requests and found that AI-authored code contained 1.7 times more problems than human-written equivalents. Excessive I/O operations were approximately 8 times more common in AI-generated pull requests [6]. The IEEE ISSRE (2025) found that AI-generated code tends to be more repetitive and structurally simpler than human code, but contains significantly more high-risk bugs [9].

B. Security Vulnerability Rate

Veracode (2025) found that 45% of coding tasks completed with AI assistance introduced at least one security vulnerability [8]. DryRun Security (2026) reported that 87% of AI-generated pull requests introduced a security issue. These rates are particularly alarming in the context of embedded software, where security vulnerabilities can translate directly into physical safety risks.

C. The Embedded Domain Specifically

RunSafe Security (2025) surveyed over 200 embedded professionals and found that 53% identified scaled-up security vulnerabilities as their primary concern with AI code, and one in three organisations had experienced an embedded cyber incident in the past year [4].

Black Duck (2025) identified what they termed the Reality Gap: 86% of engineering managers believed AI code was production-ready, while only 56% of hands-on developers agreed — a 30-point divergence that is driving premature deployment decisions [5].

D. Production Behaviour

Lightrun (2026) surveyed 200 senior engineering leaders and found that 43% of AI-generated code changes required manual debugging in production even after passing all automated quality checks. Not a single respondent described themselves as very confident in how AI code would behave once deployed [7]. This paper extends the existing body of literature by focusing specifically on system-level software — OS kernels, device drivers, firmware, and compiler toolchains — where failure consequences are categorically more severe.

III. METHODOLOGY

This study used a mixed-method research design, combining quantitative benchmarking with qualitative case study analysis. The research followed five sequential stages.

Literature Review: systematic survey of IEEE, ACM, and industry publications from 2023–2026 covering AI code quality, embedded security, and compiler correctness. Benchmarking Analysis: extraction and analysis of published performance metrics comparing AI-generated code against human-written baselines.

Case Study Analysis: structured deep-dives into four documented incidents where AI was directly involved in OS or embedded system failures or security findings.

Survey Data Processing: analysis of three large-scale industry surveys covering a combined total of over 600 embedded and DevOps engineering professionals.

Synthesis and Modelling: cross-case pattern identification feeding into the proposed governance model.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, June 2026
Impact Factor: 8.2

A. Scope

This study covers OS kernels and kernel modules, hardware device drivers, microcontroller and SoC firmware, compiler toolchains, and RTOS application code written in C, C++, or Rust. Web applications, mobile apps, and general-purpose scripts are excluded. The time period is mid-2024 through April 2026, capturing the transition from AI-as-experiment to AI-as-standard-workflow in embedded engineering.

B. Case Study Selection Criteria

Cases were selected against three criteria: (1) direct involvement of AI in embedded or OS-level software; (2) availability of a published technical post-mortem or peer-reviewed analysis; (3) clear representativeness of at least one of the three failure axes — reliability, security, or maintainability. All four selected cases satisfy all three criteria.

IV. CASE STUDIES

A. Claude's C Compiler (CCC) — February 2026

In February 2026, Anthropic researcher Nicholas Carlini documented a two-week experiment in which 16 parallel AI agents collaboratively wrote a C compiler from scratch in Rust. A compiler is the fundamental tool that translates human-readable source code into the machine instructions a processor executes. The project used approximately 2,000 AI sessions, cost around $20,000, and produced a 186,000-line codebase that was released publicly [1].

The headline claim was that CCC successfully compiled Linux kernel 6.9. Within hours of release, the first GitHub issue was filed: Hello world does not compile. On standard Fedora and Ubuntu installations, the compiler immediately failed with missing header errors because the AI had hardcoded file paths that only existed on the specific development machine [3]. GCC compiled the same program without issue.

Performance results were equally problematic. A database benchmark (SQLite) that GCC completed in 10.3 seconds took the AI compiler 2 hours and 6 minutes — a 12x regression. Optimisation flags critical for embedded timing constraints (-O2, -O3) were silently accepted but had zero effect, producing output identical to the unoptimised -O0 setting [3]. The kernel compilation claim required a further qualification: CCC could compile all source files but could not link them into a bootable binary due to incorrect ELF relocations for kernel data structures [3]. Table I summarises the benchmark comparison.

B. Linux Kernel AI Governance Crisis — 2025

The Linux kernel runs on more devices than any other OS — servers, Android phones, automotive systems, embedded industrial controllers. Its code review process is one of the most rigorous in open-source software. In July 2025, senior maintainer Sasha Levin (NVIDIA) presented AI-assisted kernel patch practices at the North American Open Source Summit [12].

This triggered a formal governance debate after it emerged that AI-generated patches were being submitted without disclosure. Maintainers identified a pattern of subtle logic errors that passed standard review — valid syntax, passing CI tests — but failed under specific hardware conditions or race scenarios. The community characterised these as code that is bad in ways humans do not typically write bad code. The outcome was a formal proposal for a mandatory Co-developed-by: AI disclosure tag. The key finding: the world's most scrutinised code review process was not reliably catching AI-characteristic correctness failures.

C. CVE-2025-37899 — Linux Kernel RCE

CVE-2025-37899 is a remote code execution vulnerability in the Linux kernel's ksmbd module — the in-kernel SMB3 file-sharing server — discovered using an AI tool (OpenAI's o3 model) [11]. The vulnerability involved a use-after-free condition in kernel session-handling logic. An unauthenticated network attacker could exploit it to achieve kernel-mode code execution on the target machine.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 11, June 2026 Impact Factor: 8.2

This case is significant on two levels. First, it confirms that AI tools have reached the capability level of discovering novel, exploitable bugs in production kernel code — not just in synthetic test environments. Second, it establishes the feedback dynamic central to this paper's security argument: if AI generates new code with the same memory-safety patterns present in its training data, the same AI discovery tools will find those AI-introduced bugs just as readily.

D. Claude Mythos — Autonomous OS Exploitation — April 2026

In April 2026, Anthropic's red team published evaluation results for Claude Mythos Preview [2]. The model was found capable of autonomously identifying and exploiting zero-day vulnerabilities across every major operating system and web browser when directed to do so.

Three specific findings are directly relevant to embedded OS engineering. First, Mythos independently discovered and exploited a 27-year-old zero-day in OpenBSD — an OS built specifically with security as its primary design goal. Second, it achieved local privilege escalation on Linux by chaining a kernel race condition with a Kernel Address Space Layout Randomisation (KASLR) bypass. Third, it constructed a working remote code execution exploit against FreeBSD's NFS server using a 20-gadget Return-Oriented Programming (ROP) chain split across multiple network packets, granting unauthenticated root access [2].

Anthropic confirmed these capabilities were not deliberately trained. They emerged as a downstream consequence of general improvements in code comprehension, reasoning, and autonomous planning. The direct implication: the same model improvements that make AI better at writing embedded OS code also make it better at attacking that code.

V. RESULTS AND DISCUSSION

A. Reliability: Contextual Brittleness

A consistent failure pattern runs across all four case studies: AI-generated system code behaves correctly under the conditions present during development, then fails when any environmental assumption changes. We term this Contextual Brittleness. In the CCC case, the compiler worked on the developer's machine but failed on every standard Linux installation. In the kernel governance case, AI patches passed CI but failed under specific hardware race conditions.

For embedded systems, where the same firmware must run reliably across hardware variants over a product lifecycle of 10–15 years, this failure mode is incompatible with production deployment without additional validation infrastructure. The Lightrun (2026) finding that 43% of AI code changes require production debugging after passing all automated checks [7] confirms this is systemic, not anecdotal.

B. Security: The Closed Exploit Loop

Three CWE classes dominate AI-generated embedded code vulnerabilities: CWE-787 (Out-of-Bounds Write / Buffer Overflow), CWE-416 (Use-After-Free), and CWE-362 (Race Condition). These three classes account for the majority of embedded security incidents [4], and all three are prevalent in the decades of legacy C/C++ code on which AI models were trained.

The Mythos evaluation establishes what this paper terms the Closed Exploit Loop: (1) AI generates embedded code containing memory-safety vulnerabilities reproduced from training patterns; (2) AI-powered security tools discover those vulnerabilities in deployed systems; (3) AI exploit frameworks construct working exploits autonomously. All three stages are confirmed operational as of April 2026. The traditional assumption that kernel vulnerabilities are difficult to discover and exploit is no longer a reliable security property.

C. Maintainability: Long-Term Cost Compounding

The CCC codebase — 186,000 lines of AI-generated Rust without persistent human architectural oversight — was described by its own author as effectively unmaintainable. External reviewers characterised its assembly output as

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, June 2026
Impact Factor: 8.2

resembling undergraduate-level compiler work. This is consistent with the Big Ball of Mud anti-pattern: locally functional code lacking global structural coherence, resistant to refactoring, and fragile under modification.

At industry scale, Black Duck (2025) found that developers spend 66% more time reviewing and correcting AI-generated code than writing equivalent code from scratch [5]. For embedded firmware that must be maintained, security-patched, and hardware-ported for a decade or more, this overhead is not a one-time cost — it compounds with every release cycle.

D. Survey Data Summary

Table II summarises the key data points from the industry surveys underpinning this analysis.

VI. HYBRID GOVERNANCE MODEL

The findings of this study do not argue for rejecting AI in embedded development — AI tools deliver genuine productivity gains for boilerplate code, test scaffolding, and documentation. The argument is for structured governance that matches the risk profile of each task to the appropriate level of human oversight. We propose a four-layer model.

A. Layer 1 — Exploration

AI may be used freely for non-safety-critical tasks: generating boilerplate code, writing unit test cases, producing documentation, and exploring design alternatives. A human developer reviews all AI output before it enters the codebase. There are no additional restrictions at this layer.

B. Layer 2 — Verification

For any AI-generated code that touches safety-relevant functions, mandatory verification steps are required before integration: (a) Hardware-in-the-Loop (HIL) testing on the actual target hardware across all supported revisions; (b) Worst-Case Execution Time (WCET) measurement on target hardware, not simulation; (c) Stack depth analysis against the actual linker script and memory map; (d) CWE-class static analysis scanning for the three dominant vulnerability classes. These checks address the Contextual Brittleness failure mode directly — the CCC compiler would have been caught at HIL testing.

C. Layer 3 — Governance Gates

AI agents with tool-use access to build systems, version control, or deployment pipelines must operate under a least-privilege permission model. Reversible actions (code edits, test runs) may proceed with single-engineer approval. Irreversible or high-impact actions (production deployments, database modifications, firmware flashes to deployed hardware) require two-engineer approval before execution. This layer prevents the class of failure observed in the Amazon March 2026 outages, where AI agents without adequate approval gates caused 12-hour production disruptions.

D. Layer 4 — Audit Trail

Every codebase should include a Software Bill of Materials (SBOM) that records which components are AI-generated and which model version produced them. When a vulnerability is disclosed — such as CVE-2025-37899 in the ksmbd module — the audit trail enables rapid identification of whether any AI-generated code in the organisation's products shares the same vulnerable pattern, enabling targeted remediation rather than full codebase audit.

The model is summarised in Fig. 1 (see Appendix). It treats AI as a capable junior engineer: fast and useful, but requiring structured supervision and incremental trust expansion. This mirrors established practice in safety engineering — NASA's independent verification and validation (IV&V), medical device FDA software review under 21 CFR Part 11, and the tool qualification requirements of DO-178C. The governance logic of those frameworks applies directly to AI agent tooling.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, June 2026
Impact Factor: 8.2

VII. CONCLUSION

This paper has presented a data-driven, case study-grounded evaluation of AI-driven development in embedded and operating systems across three failure axes. The evidence is consistent: AI accelerates development velocity but introduces systemic risks in reliability, security, and maintainability that are incompatible with safety-critical deployment without structured governance.

The four case studies provide concrete, documented evidence. CCC demonstrated that AI-generated compiler toolchains exhibit Contextual Brittleness — working in the lab while failing in the field. The Linux kernel governance crisis showed that the world's most rigorous code review process was not catching AI-characteristic correctness failures. CVE-2025-37899 confirmed that AI tools find novel, exploitable kernel vulnerabilities in production code. And the Mythos evaluation established that the Closed Exploit Loop — AI generating vulnerable code, AI finding it, AI exploiting it — is operational today, not a future projection.

The appropriate response is governance, not prohibition. The Hybrid Governance Model proposed here gives embedded engineering teams a practical framework: use AI freely at the exploration layer, enforce verification on safety-relevant code, gate destructive agent actions, and maintain full provenance through SBOM tracking. Future work should prioritise three directions: hardware-aware static analysis tools that validate AI code against specific MCU timing and memory constraints; updates to IEC 61508, ISO 26262, and DO-178C to include explicit requirements for AI-generated code fractions; and longitudinal studies measuring the real-world maintenance overhead of AI-generated embedded code over full product lifecycles.

Author

Miss. Supriya Mishra has completed his Bachelor of Science in Information Technology. She is a Software Deeveloper with around 3.5 years of experience in the IT industry.Along with working, she is also currently pursuing a Computer Applications degree from Institute of Distance and Open Learning (IDOL), Mumbai University