



**Middle East Technical University
Informatics Institute**

**COMPARATIVE ANALYSIS OF THE
COMMONLY USED CODE GENERATED
BY LARGE LANGUAGE MODELS FOR
MISRA C COMPLIANCE**

**Advisor Name: Prof. Dr. Sevgi Özkan Yıldırım
(METU)**

**Student Name: Umut Öztop
(IS)**

January 2026

**TECHNICAL REPORT
METU/II-TR-2026**



**Orta Doęu Teknik Üniversitesi
Enformatik Enstitüsü**

**BÜYÜK DİL MODELLERİ İLE
OLUŞTURULAN YAYGIN KODLARIN
MISRA C UYUMLULUĞU AÇISINDAN
KARŞILAŞTIRMALI ANALİZİ**

**Danışman Adı: Prof. Dr. Sevgi Özkan Yıldırım
(ODTÜ)**

**Öğrenci Adı: Umut Öztop
(BS)**

Ocak 2026

**TEKNİK RAPOR
ODTÜ/II-TR-2026**

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Internal Use)		2. REPORT DATE 15.01.2026	
3. TITLE AND SUBTITLE Comparative Analysis Of The Commonly Used Code Generated By Large Language Models For Misra C Compliance			
4. AUTHOR (S) Umut Öztop		5. REPORT NUMBER (Internal Use) METU/II-TR-2006-	
6. SPONSORING/ MONITORING AGENCY NAME(S) AND SIGNATURE(S) Informatics Online Master's Programme, Department of Information Systems, Informatics Institute, METU			
Advisor: Prof. Dr. Sevgi Özkan Yıldırım		Signature:	
7. SUPPLEMENTARY NOTES			
8. ABSTRACT (MAXIMUM 200 WORDS) <p>This report presents whether Large Language Models (LLMs) like ChatGPT and Gemini can generate safety-critical C code compliant with MISRA C:2012. We tasked six models with implementing CRC16 and COBS algorithms, verifying the output with PC-lint Plus and functional tests. Our results show a clear drop in compliance as algorithmic complexity increases. While models excelled at the simple CRC16 task (Gemini averaging 0.3 violations), the memory-intensive COBS task caused widespread safety failures, especially regarding single-point exit rules. We observed a distinct "compliance paradox": Claude produced the fewest violations but frequently generated broken code, whereas ChatGPT achieved 100% functional accuracy but penalized its safety score by adding unrequested complexity. Ultimately, while LLMs show promise as prototyping assistants, they cannot yet autonomously generate certification-ready embedded software.</p>			
9. SUBJECT TERMS Large Language Models (LLMs), MISRA C:2012, Static Analysis, Safety-Critical Systems, Embedded Software Engineering		10. NUMBER OF PAGES 38	

TABLE OF CONTENTS

TABLE OF CONTENTS	iv
LIST OF TABLES	vi
CHAPTER 1	1
INTRODUCTION	1
CHAPTER 2	3
BACKGROUND INFO	3
2.1 Large Language Models	3
2.2 Safety-Critical Systems and Standards.....	4
2.3 MISRA	4
2.4 Cyclic Redundancy Check	5
2.5 Consistent Overhead Byte Stuffing	6
CHAPTER 3	8
RELATED WORK	8
CHAPTER 4	11
METHODOLOGY	11
4.1 Simulation Environment.....	11
4.2 Models	11
4.3 Prompt Engineering and Constraints.....	12
4.4 Performance Metrics	14
4.5 Experimental Procedure	15
CHAPTER 5	16

RESULTS	16
Prompt 1	16
Prompt 2	20
CHAPTER 6	23
DISCUSSION	23
CHAPTER 7	26
CONCLUSION.....	26
7.1 Summary of Findings	26
7.2 Recommendations and Future Work	27

LIST OF TABLES

Table 1: Functional Correctness & Stability For Prompt 1	17
Table 2: MISRA C:2012 Compliance Summary For Prompt 1	18
Table 3: Breakdown of Specific Rule Violations For Prompt 1	19
Table 4: Functional Correctness & Stability For Prompt 2	20
Table 5: MISRA C:2012 Compliance Summary For Prompt 2	21
Table 6: Breakdown of Specific Rule Violations For Prompt 2	22

LIST OF SYMBOLS / ABBREVIATIONS

LLM	Large Language Model
CRC	The Cyclic Redundancy Check
COBS	Consistent Overhead Byte Stuffing
MISRA	Motor Industry Software Reliability Association

CHAPTER 1

INTRODUCTION

The use of Large Language Models (LLMs) in software development workflows has changed the way code is written and tested in the last few years. Tools like ChatGPT, Claude, and GitHub Copilot have made developers much more productive by letting them combine complicated logic from simple language prompts. But in areas where safety is very important, like cars, planes, and medical devices, productivity can't come at the cost of reliability. To avoid failures that could have terrible effects, these systems must follow engineering standards very closely.

MISRA C is the main rule for making sure that C-based embedded software is safe, portable, and reliable. Embedded firmware must not have any undefined behaviour and must carefully control how resources are used. This is where it differs from general-purpose software. Although LLMs are effective at producing functional logic, the fact that they are probabilistic, often causes them to produce code that resemble open-source patterns rather than the defensive programming styles that safety standards require.

So, it's becoming more and more important to find out if modern LLMs can reliably write code that meets these strict compliance standards without a lot of help from people. While prior research has examined LLM performance in general coding tasks, there is an absence of thorough investigation specifically addressing the stringent MISRA C:2012 compliance for embedded algorithms.

This study seeks to simulate and assess the efficacy of six prominent LLMs (ChatGPT, Claude, DeepSeek, Gemini, and Microsoft/GitHub Copilot) in the generation of compliant embedded C code. We compare how well these models work by looking at the outputs they give for two important embedded algorithms: CRC16-IBM and Consistent Overhead Byte Stuffing (COBS).

This study employs PC-lint Plus as a static analysis tool to identify rule infringements. The study assesses the models using two primary metrics: functional correctness (does the code operate correctly?) and compliance density (how many MISRA violations occur?). This report compares these models to rigid and specification-heavy prompts to see if LLMs can be used in security-critical embedded workflows.

CHAPTER 2

BACKGROUND INFO

2.1 Large Language Models

The Transformer, which is the foundational architecture behind modern LLMs, is a deep learning framework created to process sequential data through parallel processing. In their seminal paper, "Attention Is All You Need," Vaswani et al. define this architecture as a network "based solely on attention mechanisms, dispensing with recurrence and convolutions entirely" [1]. This innovation allows models like OpenAI's ChatGPT and Google's Gemini to recognize patterns and generate coherent text and code by predicting sequences based on vast training datasets.

LLMs don't actually know how to code, they know how to predict text. The model creates runnable scripts at speed simply by anticipating the next step in the sequence, but it lacks the situational awareness to apply deeper logic or safety protocols. Consequently, you get code that functions on the surface but is silently compromised, hiding vulnerabilities and poor standards that a human developer would have flagged immediately. In high stakes engineering fields that rely on absolute precision, this unpredictability makes LLMs risky to deploy without heavy oversight.

2.2 Safety-Critical Systems and Standards

Safety-critical systems are designed systems that could have terrible consequences if they fail or break down, such as human casualties, environmental disasters, or huge financial losses. These systems are everywhere in fields where the stakes are high, like aerospace, automotive, medical devices, railways, and nuclear energy. There are strict certification standards for software development in these fields to make sure that the software is reliable and behaves as expected. For example, avionics software must meet DO-178C, automotive systems must meet ISO 26262, and medical devices must meet IEC 62304.

2.3 MISRA

The MISRA [5] (Motor Industry Software Reliability Association) coding rules are used in the industry to meet these safety standards. There are two main types of MISRA:

- MISRA C: Its main concern is the C language (C90, C99, C11, C18). The most important changes are MISRA C:2004, MISRA C:2012, and MISRA C:2023.
- MISRA C++: This is directed towards C++. The version from 2008 (which this study uses) works with C++03. A more recent version, MISRA C++:2023, has just come out to work with modern C++17, but many tools and users are still working on it.

The two standards have the same basic idea, but they have different rules because they are written in different languages (for example, C++ has classes, templates, and inheritance, while C does not). But the main ideas—staying away from dead code, making sure type safety is strict, and controlling flow are the same for everyone.

The rules in MISRA C:2012 are divided into three main groups:

- Required: Rules that must be followed, if it can't be followed it should be documented and justified by writing.
- Advisory: Suggestions that should usually be followed but are not required to be.
- Mandatory: Rules that must be followed every time the related feature is used; there is no room for error in this type of rule.

2.4 Cyclic Redundancy Check

The Cyclic Redundancy Check (CRC) is a well-known way to find errors that is used to make sure that data is correct across digital storage and communication networks. The CRC algorithm fundamentally functions by interpreting binary data as a message polynomial and performing division by a fixed generator polynomial within a Galois Field, specifically GF(2) [2]. Table-driven methods (like the Sarwate algorithm) are often used in high-performance applications to cut down on processing time. However, embedded systems with limited memory often need a bitwise algorithmic approach to use less ROM [3]. The CRC16-IBM variant used in this study (generator polynomial $0x8005$ or $x^{16}+x^{15}+x^2+1$) makes heavy use of iterative bitwise operators like logical shifts (\ll , \gg) and exclusive-ORs (\wedge) [4]. These operations are especially important

in safety-critical firmware because using bitwise operators to change signed integers is a major cause of deviations from MISRA C compliance (for example, Rule 10.1 and Directive 4.9), which leads to behaviour that is not defined by the implementation. As a result, this algorithm is a strong test case for checking that the rules for basic type compliance and static analysis are followed [5][6].

2.5 Consistent Overhead Byte Stuffing

In serial communications, Consistent Overhead Byte Stuffing (COBS) is an algorithmic encoding method that is used to frame packets efficiently. Cheshire and Baker made COBS, which takes out a certain byte (usually zero, 0x00) from a data payload to make sure that the only way to split packets is with the delimiter. COBS makes sure that there is a steady, low overhead. Serial Line Internet Protocol (SLIP) and other such older byte-stuffing methods, can unexpectedly cause packet sizes to grow. Its operation procedure is to only add one byte for every 254 bytes of data in the worst-case expansion. Embedded systems that do not require a lot of bandwidth but require low latency are its most suitable use cases. [7].

The algorithm takes out groups of bytes that aren't zero and adds one byte that tells the program where the next zero is in the data stream. The decoder can put the payload back together by going from one zero location to the next and putting the zero bytes back where they belong.

COBS needs to be able to work with memory directly with a lot of accuracy from a software engineering point of view. This means using pointers that are hard to

understand and going through buffers again and again. The algorithm needs to be very careful when it checks its limits so that it doesn't cause buffer overflows or segmentation faults. This is because it looks for offsets in a stream of bytes. Therefore, when Large Language Models (LLMs) are coding in C, COBS are used to test its effectiveness in terms of its safety for memory and compliance with strict industry standards, such as MISRA C, especially when it comes to pointer safety and array manipulation [7].

CHAPTER 3

RELATED WORK

The usage of Large Language Models (LLMs) for software generation has become a rapidly expanding field. Early studies primarily focused on functional correctness and syntactic validity in general-purpose programming languages like Python and Java. However, as the adoption of LLMs accelerates in specialized domains, research focus has shifted toward evaluating non-functional requirements such as security, maintainability, and safety-critical compliance.

Initial assessments of AI-generated code aimed to determine if models could simply produce compiling and functionally correct solutions. Chen et al. [8] presented a comprehensive evaluation of models trained on code, utilizing benchmarks like HumanEval. Their work demonstrated that while LLMs exhibit strong performance in solving standalone programming problems, they frequently struggle with reasoning and generalization limitations. Specific commercial tools have also been subjected to scrutiny. Hansson and Ellréus [9] compared the outputs of ChatGPT and GitHub Copilot, concluding that while both tools enhance productivity, they suffer from reliability consistency issues that necessitate human oversight.

Similarly, Yetistiren et al. [10] assessed GitHub Copilot's code quality, noting that while it generates functional code, the output often requires significant refactoring to meet professional standards of readability and maintainability.

As LLMs are increasingly applied to sensitive domains, researchers have begun to evaluate the security implications of automated code synthesis. Wang and Chen highlighted that foundation models often reproduce vulnerabilities present in their open-source training data. To mitigate this, techniques such as instruction tuning have been proposed. He et al. demonstrated that aligning LLMs with security-focused instructions can reduce the density of vulnerabilities in generated code. However, security is distinct from safety. In the context of embedded systems, "safety" refers to adherence to strict determinism and the avoidance of undefined behaviours, typically enforced by standards like MISRA.

The most direct predecessor to this research is the study conducted by Umer [11], who performed a comparative analysis of five popular LLMs (ChatGPT, DeepSeek, Gemini, Meta AI, and Microsoft Copilot) for compliance with MISRA C++:2008. Umer tasked models with generating common checksum algorithms using generic prompts. The study utilized PC-lint Plus for verification and found that none of the models produced fully compliant code on the first attempt. Specifically, DeepSeek performed best with the fewest violations (13), while Meta AI produced the most (67). The study concluded that LLMs are currently unreliable for ensuring full MISRA compliance without rigorous manual review or static analysis.

While Umer’s work established a baseline for MISRA compliance in C++, there remain significant gaps in the literature regarding the C language (MISRA C:2012), which is the dominant standard for embedded firmware in the automotive and industrial sectors.

First, Umer’s study focused on the 2008 C++ standard. Modern embedded workflows have largely standardized on MISRA C:2012, which introduces different constraints regarding type safety and pointer usage that were not evaluated in the C++ study.

Second, previous studies, including Umer’s [11], utilized relatively generic prompts (e.g., "implements popular Checksum algorithms"). This approach allows the LLM to select the simplest implementation path. It remains unclear how strict functional constraints—such as requiring specific polynomial selections, prohibiting lookup tables, or mandating explicit buffer capacity checks—affect an LLM's ability to maintain compliance. It is hypothesized that increasing the cognitive load of the prompt with strict functional specifications may degrade the model's ability to adhere to safety guidelines.

Finally, the landscape of LLMs evolves rapidly. Previous baselines did not include newer high-performance models such as Anthropic Claude, which has claimed significant gains in reasoning capabilities. This study addresses these gaps by benchmarking a wider array of models against the stricter MISRA C:2012 standard using specification-heavy prompts.

CHAPTER 4

METHODOLOGY

4.1 Simulation Environment

In this study, the primary environment for code generation was the web-based interfaces of LLMs. All code generation was performed in December 2025.

For verification, the simulation environment utilized PC-lint Plus version 2.0, an industry-standard static analysis tool [12]. The tool was configured to enforce MISRA C:2012 compliance. The analysis was conducted on the same local workstation to ensure consistency.

4.2 Models

To ensure a comprehensive evaluation, six distinct LLMs were selected. The models evaluated are:

- OpenAI ChatGPT (GPT-5.2): A leading general-purpose model known for high reasoning capabilities [13].
- DeepSeek-V3.2: An open-weights model that has shown strong performance in coding benchmarks [14].
- Google Gemini 3 Pro: A multimodal model optimized for speed and efficiency [15].

- Anthropic Claude 5.5 Sonnet: A model recognized for its nuance and adherence to complex instructions (added to this study to expand upon previous benchmarks) [16].
- GitHub Copilot: A widely used AI pair programmer integrated into development environments [17].
- Microsoft Copilot: A conversational AI assistant integrated into the Microsoft ecosystem [18].

Each model was treated as a "black box," meaning no fine-tuning or internal parameter adjustments were made. Each of these models had their “memory” turned off, so the code they generated weren’t affected by the prior generations. Every generation is done in a new session. This reflects the typical usage pattern of a software engineer using these tools for assistance.

4.3 Prompt Engineering and Constraints

Unlike previous studies that utilized generic prompts (e.g., "implements popular Checksum algorithms"), this study employed specification-heavy prompts to simulate real-world embedded software requirements. The prompts imposed constraints to test the models' ability to manage cognitive load while maintaining compliance. The 2 prompts chosen with different difficulty to safe code with mind. The first algorithm is easier to implement without violating any MISRA C:2012 rules. The second algorithm is much more difficult to implement because it is more complex and it also requires pointer manipulation and buffer management which is difficult to implement while not violating any MISRA C:2012 rules.

Two specific prompts were:

Prompt 1: CRC16-IBM

“Write a C code which implements CRC16-IBM algorithm.

Polynomial: 0x8005

Initial Value: 0x0000

Input/Output Reflection: Reflected (LSB first)

Final XOR: 0x0000

Use a bitwise shift calculation, not a lookup table. Code you wrote must be fully MISRA C compliant with no violations. Provide the C code and a main function with a test vector with input "123456789" to prove correctness.”

Prompt 2: COBS Encoding

“Write a C code which implements the COBS (Consistent Overhead Byte Stuffing) encoding algorithm.

Delimiter Byte: 0x00Max Block Size: 254 bytes Buffer Safety: Explicit capacity checks required

Code you wrote must be fully MISRA C compliant with no violations.

Provide the C code and a main function with the following test vector to prove correctness:

Input: {0x45, 0x00, 0x00, 0x3C, 0x1A, 0x00, 0x40, 0x06, 0xBB, 0x00, 0x12,
0x34}

Expected Output: Verify that the encoded output has no zeros and that the offsets correctly point to the location of the removed zeros.”

4.4 Performance Metrics

The generated code was evaluated based on two primary categories of metrics:

4.4.1 Functional Correctness

Before checking for compliance, the code was compiled and executed.

- **Pass:** The code compiled without errors and the main function produced the correct test vector output (e.g., the correct CRC checksum).
- **Fail:** The code failed to compile or produced incorrect output.

4.4.2 Compliance Density

For functionally correct code, PC-lint Plus was used to identify MISRA C:2012 violations.

- **Total Violations:** The absolute count of rule violations per file.
- **Rule Distribution:** How many of the violations are “Required, Advisory, Mandatory”

4.5 Experimental Procedure

The experiment followed a structured three-step process:

1. **Generation:** Each prompt was issued to each of the 6 models. This process was repeated 3 times per model to account for variability, resulting in a total of 36 distinct source files (6 models \times 2 algorithms \times 3 iterations).
2. **Functional Verification:** Each file was compiled using a standard GCC compiler to verify syntax and logic.
3. **Static Analysis:** All files were analyzed using PC-lint Plus with strict MISRA C:2012 configuration. The resulting logs were parsed to categorize and count violations. The violations in the test code weren't included in the study because the main purpose of those is to see if the algorithm generated by the AI models to see if they were correct. Only the algorithm's code was considered.

CHAPTER 5

RESULTS

This chapter represents the findings from functional tests and static analysis of the 36 C codes generated by six different LLMs. The results are categorized by functional success, individual MISRA compliance and most common MISRA C violations in two different prompts.

Prompt 1

The first metric is to demonstrate that the LLM generated code can be compiled and if it functions correctly. For the first prompt, all of the LLMs showed high build stability. Except the GitHub Copilot, the others showed high functional reliability.

Table 1: Functional Correctness & Stability For Prompt 1

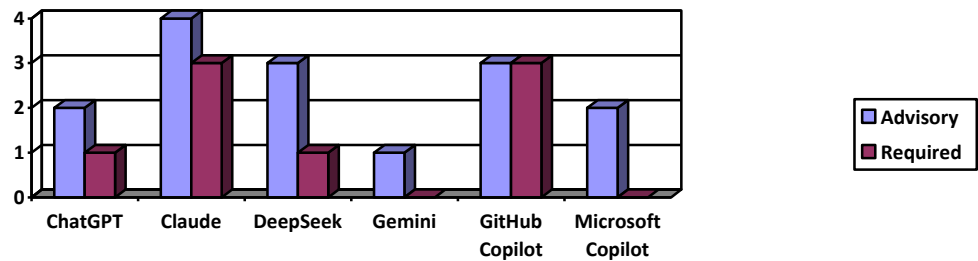
Model	Successful Builds	Functional Pass Rate (1st Try)	Functional Pass Rate (With Fixes)	Notes
ChatGPT	3/3	100%	100%	-
Claude	3/3	100%	100%	-
DeepSeek	3/3	100%	100%	-
Gemini	3/3	100%	100%	-
GitHub Copilot	3/3	33%	100%	Required 1-2 correction prompts for functionality.
Microsoft Copilot	3/3	100%	100%	-

Compliance was measured by the total number of MISRA C:2012 violations detected by PC-lint Plus. To account for variations in model verbosity and implementation scope, both total counts and averages per run were recorded.

The CRC16 implementations generally exhibited low violation counts. Gemini and Microsoft Copilot emerged as the top performers in this category, consistently producing code with near-zero violations.

Table 2: MISRA C:2012 Compliance Summary For Prompt 1

Model	Total Advisory Violations	Total Required Violations	Total Mandatory Violations	Average Advisory	Average Required	Average Mandatory	Average Violations per Run
ChatGPT	2	1	0	0.7	0.3	0	1.0
Claude	4	3	0	1.3	1	0	2.3
DeepSeek	3	1	0	1	0.3	0	1.3
Gemini	1	0	0	0.3	0	0	0.3
GitHub Copilot	3	3	0	1	1	0	2.0
Microsoft Copilot	2	0	0	0.7	0	0	0.7



To identify specific coding patterns that cause MISRA failures there are also most common violations made by different LLMs.

Table 3: Breakdown of Specific Rule Violations For Prompt 1

MISRA Rule	Category	Description	Count (All Models)	Primary Offenders
Rule 8.4	Required	Compatible declaration shall be visible.	5	The models provided a function definition but neglected to include a matching prototype in a header or at the top of the file, hindering static analysis cross-checking.
Rule 8.7	Advisory	Objects should be defined at block scope if possible.	6	Accumulator variables (e.g., uint16_t crc) or loop counters were often declared at file-scope (static) rather than locally inside the function.
Rule 10.8	Required	Essential type casting (composite expression).	2	CRC relies on shifts and XORs. Violations occurred when bitwise results (composite expressions) were cast to a wider type without explicit intermediate casting.
Rule 15.5	Advisory	A function should have a single point of exit.	5	Models used "Guard Clauses" (e.g., if (data == NULL) return 0;) at the start of the function, which violates the requirement for a single return at the end.
Rule 17.8	Advisory	A function parameter should not be modified.	2	The model modified the len or data pointer parameters directly within the loop instead of using a local pointer copy for iteration

Prompt 2

The increased complexity of pointer manipulation and buffer management led to a notable decrease in reliability. As shown only ChatGPT maintained a perfect functional record for the COBS algorithm, while other models struggled with specific logic errors or compilation issues. Also, the second prompt specifically said “encoding algorithm”, this means that ChatGPT and Microsoft Copilot misjudged or misunderstood the scope of the requested code.

Table 4: Functional Correctness & Stability For Prompt 2

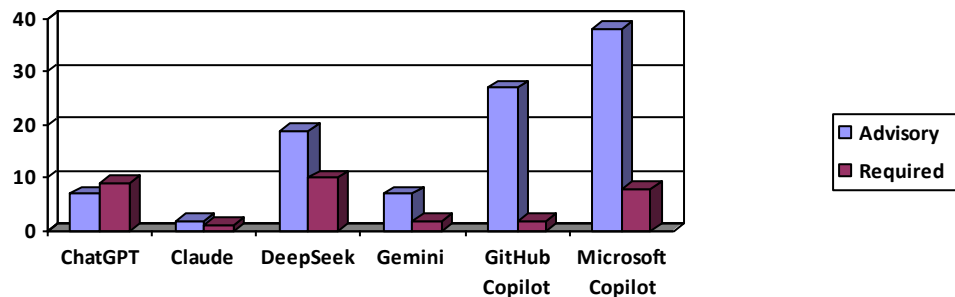
Model	Implementation Scope	Successful Builds	Pass Rate	Notes
ChatGPT	Full (Encoder + Decoder)	3/3	100%	Most reliable. Over-engineered. Implemented unrequested decoder to verify data round-trip.
Gemini	Encoder Only	2/3	100%*	Strict adherence. Run 1 failed compilation ("bool" undefined); required self-correction. After the correction, the code worked.
DeepSeek	Encoder Only	3/3	66%	Strict adherence. Run 1 failed test vector (Test code error).
GitHub Copilot	Encoder Only	3/3	66%	Strict adherence. Run 2 failed (Encoder added an extra 0).
Microsoft Copilot	Full (Encoder + Decoder)	3/3	66%	Over-engineered. Run 1 failed verification (Length mismatch, decoder added an extra 0).
Claude	Encoder Only	3/3	33%	Strict adherence but Lowest Reliability. Runs 2 & 3 failed (Added extra 0s).

Compliance was measured by the total number of MISRA C:2012 violations detected by PC-lint Plus. To account for variations in model verbosity and implementation scope, both total counts and averages per run were recorded.

The COBS algorithm presented a significant challenge. Violation counts increased sharply, particularly for models like Microsoft Copilot and DeepSeek. As noted, the results must be viewed alongside the "Implementation Scope," as models generating both an encoder and decoder produced more code, and thus more opportunities for violations.

Table 5: MISRA C:2012 Compliance Summary For Prompt 2

Model	Total Advisory Violations	Total Required Violations	Total Mandatory Violations	Average Advisory	Average Required	Average Mandatory	Total Average Violations
Claude	2	1	0	0.7	0.3	0	1.0
Gemini	7	2	0	2.3	0.7	0	3.0
ChatGPT	7	9	0	2.3	3.0	0	5.3
DeepSeek	19	10	0	6.3	3.3	0	9.7
GitHub Copilot	27	2	0	9.0	0.7	0	9.7
MS Copilot	38	8	0	12.7	2.7	0	15.4



To identify specific coding patterns that cause MISRA failures there are also most common violations made by different LLMs.

Table 6: Breakdown of Specific Rule Violations For Prompt 2

MISRA Rule	Category	Description	Total Count	Context in COBS Algorithm
Rule 15.5	Advisory	A function should have a single point of exit.	68	Models frequently used return inside if statements for error checking (e.g., buffer full), violating the "single exit" rule.
Rule 8.7	Advisory	Objects should be defined at block scope if possible.	17	Global variables were used for buffers/indices instead of passing them as function arguments.
Rule 8.4	Required	Compatible declaration shall be visible.	15	Missing function prototypes or header includes (e.g., <stdint.h> for uint8_t).
Rule 4.6	Advisory	Typography (basic types) should be used.	5	Usage of int or char instead of explicit width types like int32_t.
Rule 2.5	Advisory	A macro is not used.	5	Unused defines (e.g., MAX_BLOCK_SIZE) left in the code.
Rule 18.1	Required	Pointer arithmetic shall not result in invalid address.	3	Critical Safety Issue. Found in decoders when calculating offset locations.

CHAPTER 6

DISCUSSION

The experimental results obtained from the evaluation of 36 source files reveal several critical insights into the current state of generative AI and safety-critical software engineering. This chapter analyzes the performance disparities between models, the specific nature of MISRA C:2012 rule violations, and the practical implications of using LLMs in a certified development environment.

6.1 The Functionality vs. Compliance Trade-off

A key observation of this study is the inverse relationship between low violation counts and functional reliability in certain models. Claude exhibited the highest compliance (Avg 1.0 violations in COBS), yet had the lowest functional pass rate (33%). This suggests that Claude may be "over-optimizing" for coding standards at the expense of algorithmic logic. Conversely, ChatGPT maintained perfect functionality (100%) despite a higher raw violation count (5.3), indicating that it is a more robust tool for developers, even if the output requires manual MISRA cleanup.

6.2 Impact of Algorithmic Complexity

The jump in average violations from CRC16 (Avg 0.3–2.3) to COBS (Avg 1.0–15.3) highlights that LLMs struggle with memory-intensive tasks. While CRC16 relies on deterministic bit-shifts, LLMs performed exceptionally well here, likely because CRC implementations are ubiquitous in their training data. However, COBS requires complex pointer arithmetic and variable-length buffer management. The jump in average violations demonstrates that LLMs struggle to maintain safety constraints when the cognitive load and complexity of the algorithm increases.

6.3 The Common Violations

The most frequent violation across all models was Rule 15.5, which mandates that a function shall have a single point of exit. In the COBS dataset alone, this rule was violated 68 times.

LLMs are trained on vast repositories of C code, where the prevailing best practice is "Early Return" for error handling. While this improves readability in standard software, it is strictly prohibited in MISRA C:2012 to ensure predictable execution flow and resource cleanup. This pattern is also seen with Rule 8.7 as it was both present in the both algorithms for the most of the LLMs. Models frequently defaulted to file-scope variables for index tracking. In embedded systems, this is a safety risk as it makes the code non-re-entrant. The failure of all six models, despite explicit instructions to be "100% compliant," indicates that the models' underlying training on

common coding patterns effectively overrides their ability to follow strict, niche constraints. One of the reasons is that most of the code that is public is not MISRA compliant and because these LLMs were trained on such data they don't perform well on this topic.

6.4 Scope Creep and Evaluation Fairness

The implementation of unrequested decoders by **ChatGPT** and **Microsoft Copilot** represents a significant "Scope Creep" phenomenon. While this provided a better verification method for those models, it unfairly penalized their raw violation counts compared to models that only implemented the encoder. When normalized for complexity and code volume, ChatGPT appears significantly more capable than models that adhered to the scope. In the first glance while this looks it is better, in a strict safety-critical workflow, unrequested or excess code is considered a liability as it introduces unnecessary complexity and potential certification costs.

Conversely, Gemini, DeepSeek, and Claude demonstrated strict adherence to the prompt specifications, generating only the encoder.

CHAPTER 7

CONCLUSION

7.1 Summary of Findings

The results of this study suggest that LLMs are currently unsuitable for "Zero-Touch" code generation in safety-critical domains. However, they are highly effective as prototyping assistants.

The fact that all models achieved a high percentage of successful builds suggests that they can save significant time in the initial implementation phase. The "cleanest" model, Gemini, demonstrated that with precise prompting, an LLM can get within 3-5 violations of a perfect file. The burden remains on the human engineer to perform the final "Compliance Refactoring", specifically targeting the single exit points, explicit type casting, and variable scoping that the models consistently miss. However, this means that the human engineer should already know about the topic extensively, because they have to know which points to check or fix. But in the real world, the human coder's knowledge and their usage of LLMs to generate code is inversely proportional, known as the "Expertise Paradox" [19]. Cognitive effort required to verify AI output actually makes expert domain knowledge more critical, even as the AI handles routine labor.

Paradoxically, the advent of GenAI has made the need for human judgment and insight more essential, rather than less (Mishra, 2025). Because humans that don't have that much knowledge, need more guidance to code and the easiest, most effortless tools are the LLMs; humans that have extensive knowledge about the topic don't need that much guidance or help because they are more likely to have already done similar work. This leaves the novice in a "False Confidence Trap" where they have code that works but they lack the judgment to see why it might fail in a safety-critical environment.

According to Cui et al. [20], generative AI tools lead to productivity gains that are highest for junior developers (often 20–40% faster).

However, this rapid acceleration introduces a hidden risk: the accumulation of “Comprehension Debt” Zhang et al [21]. While juniors can now ship functional code at an expert's pace, they often do so without a foundational understanding of the logic behind it. In safety-critical environments, this debt becomes a liability; the very developers who gain the most speed from the tool are the least equipped to perform the 'Compliance Refactoring' necessary to resolve the subtle violations that LLMs consistently overlook.

7.2 Recommendations and Future Work

For engineers in the safety critical industries, LLMs should be viewed as assistants rather than autonomous coders. The high frequency of violations suggests that a human-in-the-loop process is still mandatory.

Future research should focus on "Chain of Thought" prompting, where the model is explicitly told to analyze its own code for MISRA violations before providing the final output. Additionally, investigating the compliance of specialized, fine-tuned models for embedded systems could provide a more safety-aligned alternative to the general-purpose models evaluated in this study.

REFERENCES

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998–6008, 2017.
- [2] W. W. Peterson and D. T. Brown, “Cyclic codes for error detection,” *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, Jan. 1961
- [3] Texas Instruments, “Cyclic Redundancy Check Computation: An Implementation Using the TMS320C54x,” Application Report SPRA530, Sep. 1999.
- [4] P. Koopman, “Cyclic redundancy code (CRC) polynomial selection for embedded networks,” in *The International Conference on Dependable Systems and Networks*, 2002, pp. 145–154. doi: 10.1109/DSN.2002.1028898.
- [5] MISRA, MISRA C:2012—Guidelines for the Use of the C Language in Critical Systems. Nuneaton, UK: MISRA Consortium, 2013.
- [6] L. Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. New York, NY, USA: McGraw-Hill Education, 2004.
- [7] S. Cheshire and M. Baker, “Consistent overhead byte stuffing,” *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 159–172, Apr. 1999. doi: 10.1109/90.769765.
- [8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, et al., “Evaluating Large Language Models Trained on Code,” *arXiv preprint arXiv:2107.03374*, Jul. 2021. Accessed: Dec. 15, 2025 [Online]. <https://arxiv.org/abs/2107.03374>
- [9] E. Hansson and O. Ellréus, “Code correctness and quality in the era of AI code generation: Examining ChatGPT and GitHub copilot,” Bachelor’s thesis, Dept. Comput. Sci. Media Technol., Linnaeus Univ., Växjö, Sweden, 2023. Accessed: Dec. 15, 2025 [Online]. <https://www.diva-portal.org/smash/get/diva2:1764568/FULLTEXT01.pdf>

- [10] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of GitHub Copilot's code generation," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, New York, NY, USA, 2022, pp. 62–71. doi: 10.1145/3558489.3559072.
- [11] M. M. Umer, "Comparative Analysis of the Code Generated by Popular Large Language Models (LLMs) for MISRA C++ Compliance," *IEEE Access*, vol. 13, pp. 194815–194831, 2025. doi: 10.1109/ACCESS.2025.3633086.
- [12] Vector Informatik. PC-Lint Plus . Accessed: Dec. 10, 2025. [Online]. Available: <https://pclintplus.com/>
- [13] OpenAI. ChatGPT (GPT-5.2 Model). Accessed: Dec. 10, 2025. [Online]. Available: <https://chatgpt.com/>
- [14] DeepSeek. DeepSeek-V3.2. Accessed: Dec. 10, 2025. [Online]. Available: <https://www.deepseek.com/>
- [15] Google. Gemini (2.0 Flash Model). Accessed: Dec. 10, 2025. [Online]. Available: <https://gemini.google.com/app>
- [16] Anthropic. Claude (5.5 Sonnet Model). Accessed: Dec. 10, 2025. [Online]. Available: <https://claude.ai/>
- [17] GitHub. GitHub Copilot. Accessed: Dec. 10, 2025. [Online]. Available: <https://github.com/features/copilot>
- [18] Microsoft. Microsoft Copilot. Accessed: Dec. 10, 2025. [Online]. Available: <https://copilot.microsoft.com/>
- [19] Z. Cui, M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz, "The effects of generative AI on high skilled work: Evidence from three field experiments with software developers," *SSRN Electronic Journal*, Sep. 2024. doi: 10.2139/ssrn.4945566.
- [20] Y. Zhang, "Beyond technical debt: How AI-assisted development creates comprehension debt in resource-constrained indie teams," arXiv preprint arXiv:2512.08942, 2025.

[21] P. Mishra, “The GenAI and expertise paradox: Why it makes expert work more important but harder,” *Punya Mishra’s Web*, Feb. 13, 2025. Accessed: Dec. 15, 2025 [Online]. Available: <https://punyamishra.com/2025/02/13/the-genai-and-expertise-paradox-why-it-makes-expert-work-more-important-but-harder/>