WHITE PAPER

The Limitations of Large Language Models as RTL Linting Tools for FPGA Development

A Technical Position Paper on AI-Assisted Code Review

April 2026

1. Executive Summary

Large Language Models (LLMs) have established themselves as genuinely valuable tools for RTL code generation. They can produce working Verilog and VHDL modules from natural language specifications, accelerate test bench development, and significantly reduce the time from concept to first synthesis. This paper does not dispute those capabilities. Rather, it draws a critical distinction between code generation - a creative, generative task at which LLMs excel - and code linting - a deterministic, analytical task for which they are fundamentally unsuited.

Through documented experimental evidence using Anthropic's Claude Opus 4.6, we demonstrate that when LLMs are applied to linting and structural analysis of RTL code, they produce unreliable results. In our primary test case, the model returned an incorrect analysis of a basic output bit-width - a fundamental signal property unambiguously defined in the source - while simultaneously generating multiple ancillary findings of varying validity. This behaviour creates a dangerous false confidence that could introduce errors into FPGA designs if the AI's output is trusted without exhaustive manual verification.

Our central argument is straightforward: use LLMs to write code, then use static analysis tools to lint it. These are complementary capabilities, not interchangeable ones. Organisations should integrate LLMs into their code generation workflows while maintaining deterministic EDA lint tools as the authoritative mechanism for structural and functional verification of RTL designs.

2. Introduction

2.1 Background

The FPGA development industry faces persistent pressure to reduce design cycle times while maintaining the rigorous quality standards demanded by applications in aerospace, defence, telecommunications, automotive, and medical devices. In response, engineers have increasingly adopted Large Language Models as assistants in their development workflows - and for good reason.

LLMs such as Claude, GPT-4, and Gemini have demonstrated genuinely impressive capabilities in RTL code generation. An engineer can describe a module's intended behaviour in natural language and receive a working Verilog or VHDL implementation in seconds. For common design patterns - FIFOs, state machines, AXI interfaces, SPI controllers, UART modules - the generated code is often functionally correct or very close to it, requiring only minor refinement before synthesis. This represents a meaningful productivity gain, particularly for experienced engineers who can quickly assess and correct the output, and for teams working to tight schedules on well-understood architectures.

However, this success in code generation has led to a problematic assumption: that if LLMs can write RTL code competently, they can also analyse and lint it competently. This assumption conflates two fundamentally different types of task. Code generation is a creative, probabilistic

activity - there are many valid implementations of a given specification, and the LLM needs only to produce one that works. Code linting is a deterministic, exhaustive activity - there is exactly one correct set of findings for a given rule set and body of code, and every finding must be accurate. The skills that make LLMs excellent code generators - pattern recognition, fluent text production, broad training on coding examples - are precisely the skills that make them unreliable linters.

This paper argues that the correct approach is to use each tool for what it does best: LLMs for code generation, and a dedicated static analysis tool such as Blue Pearl Visual Verification Suite for linting. These are complementary stages in a development workflow, not alternative approaches to the same problem.

2.2 Scope of Investigation

This paper documents findings from a controlled evaluation of Claude Opus 4.6 (Anthropic) when tasked with reviewing Verilog RTL code in a role-play scenario as a senior FPGA engineer. The model was presented with code samples of varying complexity and asked to identify issues, with particular attention to signal width analysis, combinational logic correctness, and synthesis-readiness.

The evaluation was conducted as part of an ongoing project to assess where AI tools may realistically contribute to hardware development workflows, and where they introduce unacceptable risk.

2.3 Why This Matters

RTL linting errors that escape into synthesis can result in hardware that behaves incorrectly in the field. Unlike software, where a bug can often be patched with an update, an error in a deployed FPGA design may require physical replacement of hardware, recall of products, or in safety-critical applications, may result in system failures with serious consequences. The bar for correctness in RTL analysis is therefore substantially higher than in typical software development.

3. Methodology

3.1 Experimental Setup

The evaluation was structured as follows:

- Model under test: Claude Opus 4.6 (Anthropic), accessed via the standard conversational interface.
- Persona instruction: The model was instructed that it was a senior FPGA engineer and was being asked to review RTL code as part of a project development process.
- Code samples: Verilog modules of varying complexity, including basic combinational logic, arithmetic operations, and signal assignments.
- Evaluation criteria: Correctness of primary analysis (e.g., output width determination), relevance and accuracy of secondary findings, and consistency across repeated evaluations.

3.2 Evaluation Framework

Each code sample was assessed against three dimensions:

Dimension	Definition	Traditional Lint Baseline
Primary Correctness	Does the tool correctly identify the specific issue under test (e.g., output bit-width mismatch)?	Deterministic tools consistently produce the correct answer for well-defined structural checks.
Secondary Validity	Are additional findings reported by the tool genuine issues or false positives?	Traditional tools have well-characterised false positive rates documented over decades of use.
Reproducibility	Does the tool produce identical results when run against the same input multiple times?	Deterministic by design. Identical input always yields identical output.

4. Evidence and Findings

Three Verilog modules of increasing complexity were submitted to Claude Opus 4.6 for review. In each case, the model was instructed that it was a senior FPGA engineer reviewing RTL code. The results are presented below, with the actual lint issues (as confirmed by Blue Pearl Visual Verification Suite and manual engineering analysis) compared against the model's findings.

4.1 Test Case 1: width_violation.v - Output Width Analysis

4.1.1 Code Under Test

A minimal five-line Verilog module performing a basic arithmetic operation:

module width_violation (input [15:0] a, b, output [15:0] answer);

assign answer = (a+b) >> 1;

endmodule

The module adds two 16-bit inputs and right-shifts the result by one. The comments in the source explicitly state that the output should be 17 bits, not 16, because adding two 16-bit numbers produces a 17-bit result. The right-shift does not reduce the required width because the full 17-bit sum must be computed before the shift; with a 16-bit output, the MSB of the sum is silently truncated, producing incorrect results when (a+b) exceeds 65,535.

4.1.2 Expected Lint Finding

A static analysis tool produces a single, unambiguous finding: the output port "answer" is declared as [15:0] (16 bits) but is assigned the result of a 17-bit addition. This is a width truncation violation. Blue Pearl flags this as a width mismatch on the continuous assignment.

4.1.3 LLM Result

Primary analysis: The model did correctly identify the loss of the carry bit, noting that the addition of two 16-bit numbers produces a 17-bit result and that the 16-bit output declaration would cause truncation. This is a positive result and demonstrates that the model can recognise well-known arithmetic width patterns.

Signal definition interpretation: The model produced an incorrect interpretation of the Verilog port declaration. The module header declares "input [15:0] a , b" which, per the IEEE 1364/1800 Language Reference Manual (LRM), defines both a and b as 16-bit inputs - the range specifier applies to all signals in the comma-separated list. The model misread this construct, demonstrating a fundamental difference between LLM-based analysis and proper static analysis tooling. Professional lint tools such as Blue Pearl Visual Verification Suite typically use the Verific front end (or equivalent LRM-compliant parser) which implements the full IEEE language standard as its parsing reference. Every declaration, expression, and construct is interpreted strictly according to the LRM specification. The LLM has no such reference - it has learned

Verilog syntax from patterns in its training data, which means its interpretation of language constructs is statistical rather than normative. When the training data contains a mixture of correct and incorrect usage examples, the model has no authoritative standard against which to resolve ambiguity. This is not a minor gap; LRM compliance is the foundation upon which all subsequent analysis depends. If the parser misinterprets a declaration, every downstream check built on that declaration - width analysis, type checking, connectivity verification - will propagate the error.

Secondary findings: Alongside the correct carry bit observation and the incorrect signal interpretation, the model generated several additional findings of questionable validity. These secondary issues were not present in the code or were mischaracterised. The engineer reviewing the output was left to separate valid findings from incorrect ones - a task that required performing the same manual analysis the model was supposed to automate.

Assessment: This test case illustrates a critical point. Even on a five-line module, the model produced a mixture of correct, incorrect, and spurious findings, all delivered with equal confidence. The incorrect signal definition interpretation is particularly telling: if the model cannot reliably parse a standard Verilog port declaration, its ability to analyse more complex constructs (parameterised widths, generate blocks, hierarchical references) cannot be trusted. On a production-scale design with hundreds of findings, the cost of manually verifying each one negates the productivity benefit of using the model. A static analysis tool, by contrast, would parse the declarations correctly and report the single width mismatch.

4.2 Test Case 2: missing_case_item.v - AXI Read Controller FSM

4.2.1 Code Under Test

A 135-line Verilog module implementing an AXI4 read transaction controller with a state machine. The code contains multiple deliberate issues that a lint tool would be expected to catch:

- A missing case item: the case statement covers fsm_clear, fsm_idle, and a default branch, but fsm_reset (2'b00) is not explicitly handled. With a 2-bit state variable and only three explicit branches (fsm_clear, fsm_idle, default), the fsm_reset state falls through to the default handler rather than being explicitly managed.
- A double semicolon on line 36 (size_reg <= 3'b000;;) which is a syntax issue that some tools flag.
- Use of blocking assignments (<=) in a combinational always @(*) block (lines 54-126), which creates a simulation vs. synthesis mismatch risk. The non-blocking operator in a combinational context can produce unexpected behaviour.
- Undriven output ports: araddr, arlen, arsize, and arburst are declared as outputs but are assigned to intermediate wires (addr_tmp, len_tmp, size_tmp, bustr_tmp) that are never connected to the output ports themselves.

4.2.2 Expected Lint Findings

Blue Pearl Visual Verification Suite identifies the missing explicit case item for fsm_reset, the undriven outputs, the non-blocking assignments in the combinational block, and width-related warnings on the len_reg_next assignment (declared as [31:0] but driven by an 8-bit source). These are all deterministic, structurally-derived findings.

4.2.3 LLM Result

Missing case item: The model's ability to identify the missing fsm_reset case item was inconsistent. This is a structural property of the case statement that requires enumerating the possible values of tc_fsm against the branches present. The model's pattern-matching approach sometimes identified this and sometimes did not, depending on the phrasing of the prompt.

Secondary findings: The model generated a mixture of valid and questionable observations. It identified some of the coding style issues but missed or mischaracterised others. Notably, the undriven output ports - which represent a functional defect that would cause the synthesised design to malfunction - were not consistently flagged.

Assessment: This module represents a realistic, mid-complexity RTL design. The LLM's performance was unreliable: it found some issues, missed others, and in some runs generated findings that were not present in the code. The inconsistency across runs is itself disqualifying - a lint tool must produce the same results every time.

4.3 Test Case 3: unreachable.v - OpenRISC Exception Handler FSM

4.3.1 Code Under Test

A 603-line module derived from the OpenRISC 1200 processor's exception handling logic. This is a complex, real-world design with conditional compilation (`ifdef directives), a multi-state FSM (seven states: IDLE through FLU6), casex statements with priority-encoded exception triggers, and deliberate bugs inserted for testing:

- Unreachable state FLU6: In state FLU5 (line 586), the next state is unconditionally assigned as FLU5 itself (state <= #1 FLU5). The correct logic - which would transition to either IDLE or FLU6 based on pipeline status - is commented out (lines 577-584). This means state FLU6 can never be reached, and the FSM will hang permanently in FLU5 once it enters that state.
- Dead code in FLU6: The entire FLU6 state handler (lines 590-597) is dead code because the state is unreachable. Additionally, the self-loop within FLU6 (line 596) is also commented out, meaning that even if FLU6 could be reached, it would not behave correctly.
- Missing #1 delay on some assignments: Lines 516 and 523 use except_type <= (without #1) while all other assignments in the same block use except_type <= #1, indicating an inconsistency.

4.3.2 Expected Lint Findings

A static analysis tool performing reachability analysis on the FSM state graph would identify that FLU6 has no incoming transitions and is therefore unreachable. It would flag the FLU6 handler as dead code. It would also flag the FLU5 self-loop as a potential hang condition (a state with no exit path). These findings require constructing the state transition graph and performing graph reachability analysis - a deterministic, algorithmic task.

4.3.3 LLM Result

Primary finding: The model correctly identified the critical FLU5 deadlock as the most serious issue. It accurately described the self-loop at line 586 (state <= #1 FLU5), recognised that the correct exit logic was commented out at lines 577-584, and understood that the FSM would hang permanently once FLU5 was entered, freezing the pipeline. It also correctly identified that FLU6 contained the proper exit logic but had its own fragility due to the commented-out self-loop at line 596, noting that without it, the state would rely on latch inference rather than explicit assignment.

Secondary findings: The model identified several additional genuine issues. It flagged the missing #1 delay on the except_type assignments at lines 516 and 523 (TRAP and SYSCALL), correctly noting these would cause simulation mismatches. It identified redundant ternary expressions where multiple branches resolved to the same value (e.g., eear <= #1 ex_dslot ? ex_pc : ex_pc on line 456). It spotted that delayed_tee was declared and computed but never read, suggesting a functional bug in tick exception gating. It also raised the well-known concerns around synopsys full_case parallel_case pragmas causing simulation/synthesis mismatches.

Assessment: This is the strongest result of the three test cases. The model correctly identified the primary critical defect and produced several valid secondary findings. However, this result must be interpreted carefully. The FLU5 bug was made highly visible by explicit comments in the source code stating "To Fix Uncomment Section below" and "To fix comment it" - these are strong textual signals that an LLM is well-equipped to recognise. The model’s success here may owe more to its ability to read and interpret comments and recognise patterns of commented-out code than to genuine structural analysis of FSM state transitions. The question is whether the model would have identified the deadlock in the absence of those comments, with only the code itself to work from. A static analysis tool does not read comments; it constructs the state transition graph and identifies unreachable states and hang conditions algorithmically, regardless of whether the code is commented or not.

4.4 Summary of Results

Test Case	Primary Defect	Blue Pearl Result	LLM Result
width_violation.v	Output width truncation (16-bit output for 17-bit sum)	Correctly identified as width mismatch on continuous assignment.	PARTIAL. Correctly identified carry bit loss, but also generated spurious secondary findings not present in the code.
missing_case_item.v	Missing explicit case for fsm_reset; undriven outputs	All structural defects identified deterministically.	INCONSISTENT. Some findings correct, others missed or hallucinated. Results varied between runs.
unreachable.v	Unreachable FSM state (FLU6); FSM hang in FLU5	State reachability analysis identifies FLU6 as unreachable and FLU5 as a hang state.	Correctly identified FLU5 deadlock and multiple valid secondary findings. However, source comments explicitly described the bug, aiding detection.

The results across the three test cases reveal a nuanced picture. The LLM is capable of identifying certain categories of defect, particularly when strong textual cues are present in the source (comments describing bugs, well-known coding patterns). However, its performance is unreliable on fundamental structural analysis tasks such as signal width parsing, and its tendency to generate spurious findings alongside valid ones creates a verification burden that undermines its value as a lint tool. A static analysis tool produces correct, deterministic, reproducible findings on every run, with no false positives from pattern-matching against training data.

4.5 The False Confidence Problem

The most concerning aspect of the observed behaviour is not the individual errors themselves - any tool can have limitations - but the manner in which errors were presented. The model delivered its incorrect analyses with the same confident, authoritative tone as its correct observations. There was no hedging, no confidence score, no indication that the model was uncertain about its answer.

This creates what we term the "false confidence problem": because LLMs generate fluent, authoritative-sounding text regardless of the accuracy of their underlying reasoning, the output appears more reliable than it actually is. In a time-pressured development environment, an engineer may be inclined to accept the model’s analysis at face value, especially when it is presented alongside several other plausible-looking findings.

4.6 The Compensatory Findings Pattern

A notable pattern emerged during the evaluation: when the model's primary analysis was incorrect, it tended to generate a larger number of secondary findings. We term this the "compensatory findings pattern" - the model appears to be pattern-matching against common Verilog issues it has encountered in training data, rather than performing genuine structural analysis of the specific code under review.

This behaviour is consistent with how LLMs operate: they predict probable text sequences based on patterns in their training corpus. When asked to review Verilog code, the model draws upon patterns from thousands of code reviews, lint reports, and engineering discussions. It can reproduce the form of a code review - the structure, the language, the categories of issues typically raised - without performing the substance of the analysis.

5. Root Cause Analysis

5.1 Probabilistic vs. Deterministic Analysis

The fundamental mismatch between LLMs and RTL linting lies in the nature of the analysis required. RTL linting is an inherently deterministic task: given a set of rules and a body of code, there is exactly one correct set of findings. Traditional lint tools achieve this by constructing a complete parse tree of the HDL code, building a signal connectivity graph, and systematically applying rule checks against the graph.

LLMs do not construct parse trees or signal graphs. They process text as sequences of tokens and generate responses based on statistical patterns learned during training. While this approach can produce impressively accurate results for many natural language and software engineering tasks, it is fundamentally unsuited to tasks that require exact structural analysis.

5.2 Specific Failure Modes

Bit-width propagation: Correctly determining the width of a signal in Verilog requires tracing assignments, operations, and port connections through the design hierarchy. An LLM may recognise common patterns (e.g., adding two N-bit numbers produces an N+1 bit result) but cannot reliably trace arbitrary signal paths, especially in designs with parameterised widths, generate blocks, or complex concatenation and slicing operations.

Clock domain crossing analysis: Identifying clock domain crossings requires building a complete clock tree for the design and mapping every register to its clock domain. This is a graph-theoretic problem that cannot be solved by text pattern matching.

Combinational loop detection: Finding combinational loops requires cycle detection in a directed graph of combinational assignments. LLMs have no mechanism to construct or traverse such graphs.

Synthesis vs. simulation mismatches: Detecting constructs that simulate correctly but synthesise differently requires knowledge of specific synthesis tool behaviours, combined with structural analysis of the code. LLMs may flag some well-known examples from their training data, but cannot reliably identify novel instances.

5.3 The Absence of Self-Verification

A critical distinction between LLMs and traditional lint tools is the absence of internal verification. When a lint tool reports a bit-width mismatch, it has computed the widths on both sides of the assignment by walking the expression tree. The finding is a direct consequence of the analysis. When an LLM reports a width issue, it has generated text that describes a width issue - but there is no underlying computation verifying that the issue actually exists in the code.

The model has no mechanism to check its own work. It cannot re-parse the code, re-evaluate an expression, or verify that its stated conclusion follows from the actual code. This is not a

limitation that can be resolved by better prompting or larger models; it is an architectural property of transformer-based language models.

6. Comparative Analysis: LLMs vs. Traditional Lint Tools

Capability	Traditional Lint Tool	LLM (Claude Opus 4.6)
Signal width analysis	Exact. Computed from parse tree.	Approximate. Inferred from text patterns. Demonstrated errors on basic cases.
Clock domain crossing	Complete graph-based analysis of clock tree and register domains.	May flag well-known CDC patterns but cannot construct clock graph.
Combinational loop detection	Cycle detection on signal dependency graph.	No graph construction capability. May identify obvious cases only.
Latch inference	Structural analysis of incomplete case/if statements.	Pattern matching on common coding errors. May miss novel constructs.
Synthesis pragmas	Tool-specific rule sets, updated per release.	Training data dependent. May not reflect current tool versions.
Reproducibility	Deterministic. Same input always yields same output.	Non-deterministic. Varying results across runs.
False positive rate	Well-characterised and documented.	Uncharacterised. No systematic data available.
Audit trail suitability	Yes. Deterministic results support compliance.	No. Non-deterministic output unsuitable for regulated workflows.
Processing speed	Seconds to minutes for large designs.	Seconds per module, but no batch or hierarchical capability.
Design hierarchy	Full elaboration and cross-module analysis.	Single-module context window. Cannot trace inter-module signals.

7. Implications for Industry Practice

7.1 Safety-Critical Applications

In domains governed by functional safety standards such as IEC 61508, DO-254, and ISO 26262, the use of development tools is itself subject to qualification requirements. Tools must demonstrate known, repeatable behaviour, and their limitations must be documented and accounted for in the safety case. LLMs, by their non-deterministic nature, cannot currently meet these qualification requirements.

Any organisation subject to these standards should explicitly exclude LLM-based linting from their tool qualification strategy and ensure that engineers understand that AI-generated code reviews do not constitute verified analysis.

7.2 Commercial and Industrial Applications

Even outside safety-critical domains, the reliability concerns documented in this paper should give pause. An incorrect width analysis that reaches synthesis could result in data corruption, protocol non-compliance, or intermittent failures that are extremely difficult to diagnose in the field. The cost of debugging such issues in deployed hardware typically exceeds the cost of proper linting by orders of magnitude.

7.3 The Human Factor

There is an additional risk that merits consideration: the potential for automation complacency. As engineers become accustomed to AI-assisted review, there is a natural tendency to reduce the intensity of manual review. If the AI tool has an uncharacterised error rate - as demonstrated in this evaluation - this complacency can lead to a net reduction in design quality, even if the AI occasionally catches genuine issues that a human reviewer might miss.

8. Where AI Excels: Code Generation and the Recommended Workflow

It is essential to emphasise that this paper is not an argument against using LLMs in FPGA development. On the contrary, LLMs have demonstrated clear, measurable value in RTL code generation - and teams that are not leveraging this capability are leaving productivity gains on the table.

8.1 LLMs as Code Generators

LLMs excel at code generation because it plays directly to their architectural strengths. Generating a Verilog module from a natural language specification is a creative, pattern-based task: the model draws on its extensive training corpus of HDL examples to produce code that matches the described intent. There are many valid ways to implement a given specification, and the model needs only to produce one that is functionally sound. This is fundamentally different from the exhaustive, single-correct-answer nature of linting.

In practice, LLMs have proven effective at generating working implementations of common design patterns including FIFO buffers, state machines, bus interfaces (AXI, Wishbone, APB), serial communication controllers (UART, SPI, I²C), clock dividers, debounce circuits, and parameterised arithmetic units. For experienced engineers, this can reduce initial coding time from hours to minutes, allowing more time to be spent on architecture decisions, verification, and optimisation.

8.2 The Recommended Workflow: Generate, Then Lint

The optimal workflow combines the strengths of both approaches in sequence:

- Step 1 - Generate: Use an LLM to produce initial RTL code from a specification or natural language description. Review the generated code for architectural intent and functional correctness at a high level.
- Step 2 - Lint: Run the generated code through a deterministic static analysis tool such as Blue Pearl Visual Verification Suite. This catches the structural issues (width mismatches, clock domain crossings, latch inference, combinational loops) that the LLM cannot reliably detect.
- Step 3 - Iterate: Use the LLM to assist with fixing lint findings where appropriate, particularly for refactoring or restructuring code to address style and structural issues. Run the lint tool again to verify the fixes.
- Step 4 - Verify: Proceed to simulation and formal verification with confidence that the code has passed deterministic structural checks.

This workflow treats code generation and code analysis as complementary stages, not interchangeable activities. The LLM accelerates the creative work; the static analysis tool provides the rigorous verification. Neither replaces the other.

8.3 Additional Areas of LLM Value

Beyond code generation, LLMs offer genuine value in several supporting activities:

- Test bench generation: Producing initial test bench structures, stimulus patterns, and self-checking test frameworks. The LLM's ability to generate comprehensive corner-case stimulus from a specification description is particularly useful.
- Documentation: Generating module descriptions, port tables, interface documentation, and design rationale from RTL source code. The engineer verifies accuracy, but the time saving is substantial.
- Coding style and convention review: Checking naming conventions, comment quality, and adherence to team style guides - areas where false positives carry minimal risk.
- Learning and training: Junior engineers can use LLMs as an interactive reference for HDL syntax, design patterns, and best practices, provided they understand the distinction between generation and analysis.

In each of these cases, the AI's output is a productivity accelerator that still requires human judgement. The critical point is that none of these applications substitute for static analysis linting, which remains the domain of purpose-built EDA tools.

9. Recommendations

Based on the evidence presented in this paper, we make the following recommendations:

1. **Embrace LLMs for code generation.** They are proven productivity tools for producing initial RTL implementations, test benches, and supporting infrastructure. Teams not using them for generation are missing a genuine advantage.
2. **Always lint LLM-generated code with a static analysis tool.** Treat generated RTL exactly as you would code written by a capable but fallible colleague: it needs the same rigorous lint and verification flow as any hand-written code. This is not optional.
3. **Do not use LLMs as a replacement for deterministic lint tools.** The architectural limitations documented here are not resolved by prompt engineering, model scaling, or fine-tuning. An LLM reviewing code is generating plausible commentary, not performing structural analysis.
4. **Explicitly exclude LLM-based linting from safety-critical tool chains.** Document this exclusion in the project's tool qualification plan. LLM-assisted code generation may still be permissible, provided the generated code passes the full qualified verification flow.
5. **Educate engineering teams on the generation–analysis distinction.** Ensure that all engineers understand that an LLM's ability to write correct code does not imply an ability to verify code correctness. These are fundamentally different tasks.
6. **Monitor developments in hybrid approaches** that combine LLM front-ends with formal analysis engines. Such tools may eventually bridge the generation–analysis gap, but should be evaluated against the same rigorous criteria applied to any other EDA tool.

10. Conclusions

The evidence gathered during this evaluation leads to a clear and actionable conclusion: LLMs are powerful code generation tools and unreliable code analysis tools. These two capabilities are not correlated - excelling at one does not imply competence at the other - and conflating them introduces unacceptable risk into FPGA development workflows.

The specific limitations documented in this paper are architectural, not incidental:

- LLMs generate text that resembles code analysis, rather than performing actual structural analysis of the design. They predict what a lint report should look like, without executing the underlying computations.
- They lack internal verification mechanisms, meaning they cannot detect or correct their own errors. When the model gets a bit-width wrong, nothing in its architecture signals that the answer is incorrect.
- Their non-deterministic nature makes them unsuitable for any workflow requiring repeatability and audit trails - a hard requirement in regulated industries.
- The false confidence problem - presenting incorrect analysis with the same authoritative tone as correct analysis - creates a direct risk of undetected errors reaching synthesis.

The correct response to these findings is not to abandon LLMs, but to deploy them where they are strong. Use LLMs to accelerate code generation, test bench development, and documentation. Then use a deterministic static analysis tool such as Blue Pearl Visual Verification Suite to perform the rigorous structural verification that the design requires. This is not a compromise; it is the optimal workflow, combining the productivity benefits of AI-assisted generation with the reliability guarantees of proven EDA tooling.

The question is not whether to use AI in FPGA development. It is where to use it. The answer, supported by the evidence in this paper, is straightforward: generate with AI, lint with static analysis.

Model Evaluated: Claude Opus 4.6 (Anthropic)
Date: April 2026