AI Coding Agents Need Better Compiler Remarks

Akash Deo
Northwestern University
Evanston, Illinois, USA

Simone Campanoni
Northwestern University
Evanston, Illinois, USA

Tommy McMichen
Northwestern University
Evanston, Illinois, USA

Abstract

Modern AI agents optimize programs by refactoring source code to trigger trusted compiler transformations. This preserves program semantics and reduces source code pollution, making the program easier to maintain and portable across architectures. However, this collaborative workflow is limited by legacy compiler interfaces, which obscure analysis behind unstructured, lossy optimization remarks that have been designed for human intuition rather than machine logic. Using the TSVC benchmark, we evaluate the efficacy of existing optimization feedback. We find that while precise remarks provide actionable feedback (3.3× success rate), ambiguous remarks are actively detrimental, triggering semantic-breaking hallucinations. By replacing ambiguous remarks with precise ones, we show that structured, precise analysis information unlocks the capabilities of small models, proving that the bottleneck is the interface, not the agent. We conclude that future compilers must expose structured, actionable feedback designed specifically for the future of autonomous performance engineering.

1 Problem Statement

Large Language Models (LLMs) are rapidly evolving from simple code generators into autonomous performance engineers. Recent work demonstrates that these agents can successfully perform complex, source-level refactoring—such as loop splitting and array privatization—to assist compilers in generating highly optimized binaries. However, a critical bottleneck restricts the efficacy of this collaboration: the *limited insight* provided by traditional compiler interfaces.

Currently, coding agents operate with an incomplete view of the compiler's analysis. While compiler remarks are generally trustworthy, they are often *vague and opaque*, providing little guidance on how to resolve a failed optimization. The core issue is not a lack of agent capability, but a lack of *actionable signal*. To effectively close the optimization loop, an agent requires feedback that is both **insightful** (*why* it failed) and **prescriptive** (*where* and *how* to fix it).

Such remarks effectively guide the agent to a correct solution. On the contrary, many remarks are vague, identifying a failure without providing a path forward. Worst of all, some remarks are actively detrimental, introducing ambiguous information that misleads the agent into hallucinating invalid code modifications or breaking program semantics. To unlock the next era of *AI with Compilers*, tools must focus on the quality of remarks, exposing insightful analysis results with prescriptive feedback designed for agentic consumption.

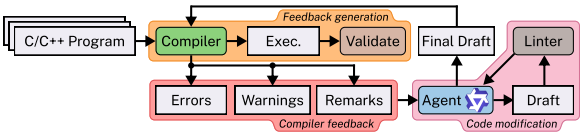

Figure 1. Agentic workflow used in our evaluation.

2 Related Works

LLM-Driven Code Transformation. Systems like LLM-Vectorizer [3] and Astra [4] bypass compiler safety analyses using low-level intrinsics. VecTrans [5] introduced an iterative loop where an agent is guided by compiler feedback to enable auto-vectorization via source-level transformations. However, VecTrans treats the compiler's output as a fixed, immutable signal. This paper shifts the focus to the *quality* of that signal. We demonstrate that existing feedback is a fundamental bottleneck for AI coding agents and argue for a co-designed interface that exposes deep analytical insights for effective refactoring.

3 Design and Methodology

We propose an agentic workflow (Figure 1) where an engineer agent is provided with a C/C++ program and comprehensive compiler feedback (warnings, errors, and optimization remarks[1]) and is tasked with performing source-level transformations to enable auto-vectorization. We utilize a single-pass evaluation to isolate the impact of diagnostic quality on success rates, rather than measuring trial-and-error in an iterative flow. The agent refactors based on compiler feedback, with up to three attempts to fix syntax errors. We validate by inspecting optimization records for vectorization success and checking semantics via differential testing[2].

4 Evaluation

We evaluate Qwen2.5-coder (8B parameters) on TSVC [2], a benchmark suite consisting of 151 loops designed to test compiler auto-vectorization. For each loop, we run the workflow under four configurations: **Clang** 21.8 and **Intel** 2025.3 compilers, both **with** and **without** optimization remarks. We also vary sampling temperature ($T = \{0.2, 0.8, 1.2\}$) to provide insight into the 'creativity' needed to enable vectorization, where a higher temperature allows for more stochastic explorations. We run 100 trials per loop for each configuration.

[1] `clang -fsave-optimization-record` and `icx -qopt-report=max`.
[2] While prior work uses Alive2 [1] for formal translation validation, we utilize differential testing as a practical baseline for detecting broken semantics.

Our primary goal is to measure how the presence and quality of these remarks influence the agent's success.

Remarks Improve Success. Table 1 shows compiler feedback as the determinant factor for success. Without remarks, vectorization rates are negligible (<1.5% for Clang, <3.7% for Intel) across all temperatures. Remarks act as a significant performance multiplier. At $T = 0.8$, providing remarks increases Clang's success rate from 0.80% to 2.68% (3.3× increase) and Intel's from 2.38% to 6.95% (2.9× increase). Notably, at $T = 0.2$, the difference in success rate with Intel remarks is much greater than with Clang remarks, suggesting that higher-fidelity diagnostics[3] reduce the creativity burden on the agent, enabling success at low temperatures.

The Benefit of Precision and Cost of Ambiguity. For each remark, we aggregate success rates across all benchmarks where that remark appears, shown in Table 2, which confirms that the utility of feedback is not uniform. Precise remarks provide massive gains. For example, identifying an output dependence or anti dependence results in success rate deltas of +15.89% and +8.64%, respectively. Conversely, vague remarks can be actively *detrimental*. Clang's NonReductionValueUsedOutsideLoop remark consistently results in negative success rates (up to −2.42%), indicating the agent is more likely to succeed if the feedback is omitted. In these cases, the ambiguous signal triggers semantic hallucinations where the agent attempts to satisfy a poorly communicated constraint by breaking program logic. For example, agents would often incorrectly break loop-carried dependencies when inserting temporaries.

Temperature Sensitivity. Success rates generally scale with temperature, confirming that satisfying complex vectorization constraints often requires the stochastic 'creativity' found at $T = 1.2$, whereas those same solutions are pruned at low temperatures. However, certain structural fixes, such as resolving Intel's Multiple Exits remark, peak at $T = 0.2$ (+10.61%) and degrade as temperature increases.

Precise Remarks To investigate how existing remarks can be improved, we hand-write precise remarks, providing detailed insights for why vectorization failed. In particular, we expose the underlying dependence analysis instead of producing a detrimental "unsafe dependent memory operations in loop" remark in Clang. For instance, we provide "assumed write-after-read dependence between a[i+1] [tsvc.c:1148:13] and a[i] [tsvc.c:1149:27]. suggestion: consider using a temporary to store a[i+1]." With precise remarks, success rates drastically improve: identifying write-after-read dependencies achieves +52% at T=0.2, while read-after-write shows +49% at T=0.8. These gains demonstrate that exposing precise dependence information enables agents to reliably apply transformations. Write-after-write shows inconsistent results across temperatures, suggesting that dependence information alone is insufficient. Additional analysis may be needed to make this remark actionable.

[3]This aligns with the general consensus that the Intel compiler provides more comprehensive and actionable optimization reports than Clang.

Compiler	Config	T=0.2	T=0.8	T=1.2
Clang	No Remarks	0.20%	0.80%	1.45%
Clang	Remarks	0.64%	2.68%	3.93%
Intel	No Remarks	1.10%	2.38%	3.67%
Intel	Remarks	4.59%	6.95%	7.83%

Table 1. Success rate of vectorization using different compilers with varying temperatures over 100 trials. Benchmarks that can be vectorized without refactoring are excluded.

	Remark	T=0.2	T=0.8	T=1.2
Intel	Output Dependence	+3.94	**+15.89**	+5.87
	Anti Dependence	+1.19	**+8.64**	+7.62
	Multiple Exits	**+10.61**	+4.72	+3.21
	Flow Dependence	+1.02	+1.92	**+2.55**
	Outer Loop	-1.46	-1.31	-1.28
	Loop Control Var	-1.56	-2.61	-2.96
	Function Call	-1.56	-2.78	-3.63
	Seems Inefficient	-1.56	-3.11	-4.13
Clang	ArrayBounds	**+5.44**	+4.39	+4.87
	EarlyExit	-1.56	**+2.89**	-0.63
	UnsafeDependency	-1.44	-0.53	-0.17
	Libcall/Instr	-1.56	-1.86	-3.63
	NonReductionValue	-1.53	-2.20	-2.42
Precise	ReadAfterWrite	0.00	**+49.00**	+35.00
	WriteAfterRead	**+52.00**	+35.00	+11.20
	WriteAfterWrite	0.00	**+8.00**	-8.00

Table 2. Difference in success rate with and without remarks, aggregated by remark type.

5 Discussion and Conclusion

Agents thrive on precise, data-flow-level signals, such as explicit dependencies with source-level debug information, while collapsing under ambiguous structural warnings. To resolve current limitations, agents require structured access to the compiler's internal analysis, such as dependence and alias analysis. Exposing *why* the transformation failed allows the agent to distinguish between genuine data hazards and conservative static approximations.

We demonstrate that the bottleneck for AI-driven optimization is not model capability, but the opacity of traditional compiler interfaces. Precise diagnostics act as a performance multiplier, whereas vague signals provoke hallucinations that break semantics. By evolving from purely descriptive

observations to structured, prescriptive analysis, we transform the AI agent from a stochastic code generator into a reliable performance engineer capable of navigating complex, semantics-preserving optimizations.

AI Use Statement

AI tools were used for drafting and editing the paper along with implementing parts of the experimental artifact.

References

[1] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI*. https://doi.org/10.1145/3453483.3454030

[2] Saeed Maleki, Yaoqing Gao, María J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 372–382. doi:10.1109/PACT.2011.68

[3] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K. Lahiri. 2025. LLM-Vectorizer: LLM-Based Verified Loop Vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas NV USA, 2025-03). ACM, 137–149. doi:10.1145/3696443.3708929

[4] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A Multi-Agent System for GPU Kernel Performance Optimization. arXiv:2509.07506 [cs] doi:10.48550/arXiv.2509.07506

[5] Zhongchun Zheng, Kan Wu, Long Cheng, Lu Li, Rodrigo C. O. Rocha, Tianyi Liu, Wei Wei, Jianjiang Zeng, Xianwei Zhang, and Yaoqing Gao. 2025. VecTrans: Enhancing Compiler Auto-Vectorization through LLM-Assisted Code Transformations. (2025). arXiv:2503.19449 [cs.SE] https://arxiv.org/abs/2503.19449