Building a C compiler with Claude: moving beyond the snippet

We usually treat LLMs as tools for the easy stuff: boilerplate, unit tests, or maybe a quick Python script to rename files. But building a compiler? That is different. A compiler is intolerant of ambiguity; it is the "final boss" of exactness. One missing semicolon, one misaligned register, and the whole system collapses.

Anthropic's engineering team recently documented their process of building a C compiler using Claude. It is a fascinating read, not because the resulting compiler is groundbreaking technology, but because it breaks the stereotype that LLMs struggle with rigorous, multi-stage logic.

The shift from syntax to architecture

The key takeaway isn't that Claude knows C (it obviously does). It is how the engineering workflow changes when the model handles the implementation details. You stop worrying about syntax or memory allocation, and you start worrying about the design of the Abstract Syntax Tree (AST) and the code generation strategy.

The post highlights that success didn't come from a single magical prompt. It came from treating the project as a system: breaking it down into the lexer, the parser, and the code generator, and iterating on each component. The model acted less like a text generator and more like a pair programmer that had memorized the C specification.

Bridging the gap between probabilistic and deterministic

This experiment highlights the friction between the probabilistic nature of LLMs and the deterministic requirements of systems programming. A compiler cannot "hallucinate" a valid assembly instruction; it must be mathematically correct. The team found that by constraining the scope of each step—asking the model to solve small, well-defined problems rather than the whole compiler at once—they could effectively "tame" the randomness.

This is the essence of effective agentic workflows. We are moving past the era where AI is just for "toy apps." The bottleneck is no longer the model's ability to write complex code; it is our ability to architect systems that can consume that code safely.

Why this matters for everyday engineering

This signals a maturity in "Agentic Engineering." If a model can maintain enough context and logical coherence to build a functional compiler—a system where a single error anywhere breaks everything—then the argument that "AI can't handle complexity" is getting weaker.

It forces us to ask: if the AI handles the implementation, what is left for us? The answer, clearly, is architecture. The value shifts from knowing *how* to write a specific sorting algorithm to knowing *why* that algorithm fits into the broader system.

Contributor: Alessandro Linzi