From Natural Language to Hardware: The Idea2Circuit Pipeline for Automated Circuit Synthesis via LLM-Guided Code Generation and Iterative Compilation

Shyamal Suhana Chandra
Sapana Micro Software
Pittsburg, KS 66762
Email: sapanamicrosoftware@gmail.com

***Abstract*—We present Idea2Circuit, an automated pipeline that transforms natural-language descriptions of computational ideas into hardware circuit schematics targeting ASIC, FPGA, TPU, QPU, OPU, LPU, and GPU architectures. The system employs a multi-stage architecture comprising (i) LLM-based C code generation from unstructured text, (ii) recursive validation and error-corrected compilation using gcc with strict warning flags, (iii) static heuristic test generation at a density of twenty tests per source line across five categories, (iv) software design pattern conformance checking, and (v) hardware synthesis via the Flux compilation API with graceful degradation to a mock circuit generator. The pipeline achieves correct-by-construction C code within a mean of 2.4 validation iterations on a corpus of fifteen benchmark ideas, with a 93% success rate for FPGA-targeted circuits. This paper details the system architecture, algorithmic components, design decisions, and empirical results, and discusses limitations and directions for future work in LLM-driven hardware synthesis.**

***Index Terms*—Hardware synthesis, large language models, code generation, circuit design, FPGA, ASIC, automated compilation, LLM-guided programming**

I. Introduction

The translation of high-level human intent into executable hardware descriptions remains a fundamental challenge in electronic design automation (EDA). Traditional hardware description languages (HDLs) such as VHDL, Verilog, and SystemVerilog require specialized expertise and exhibit a steep learning curve that places hardware design beyond the reach of most software engineers [13]. Concurrently, large language models (LLMs) have demonstrated remarkable proficiency in generating software code from natural-language prompts [1], but their application to hardware synthesis remains nascent.

Idea2Circuit addresses this gap by introducing an end-to-end pipeline that converts natural-language ideas into synthesizable circuit schematics through an intermediate C-code representation. The key insight is that modern high-level synthesis (HLS) tools can compile well-structured C programs into hardware descriptions [2], and that LLMs can generate such C programs when guided by an appropriately structured prompt and iterative error-correction loop.

The principal contributions of this work are:

1) A complete six-stage pipeline architecture for natural-language-to-hardware synthesis, comprising generation, validation, testing, pattern analysis, and compilation stages.
2) A recursive validation algorithm that combines static pre-validation with gcc compilation and LLM-driven error correction, achieving high first-pass correctness.
3) A static heuristic test-generation methodology that produces twenty tests per source line across five orthogonal categories without requiring runtime execution.
4) A comprehensive design-pattern conformance checking system that evaluates ten software engineering patterns in generated hardware-oriented C code.
5) An empirical evaluation on fifteen benchmark ideas spanning five hardware target architectures.

The remainder of this paper is organized as follows. Section II provides background on LLM code generation, high-level synthesis, and prior work. Section III describes the system architecture and its components. Section IV details the core algorithms. Section V presents benchmark results. Section VI discusses architecture choices and trade-offs. Section VII analyzes results. Section VIII outlines future work. Section IX concludes. Section IX concludes and references follow.

II. Background and Related Work

A. LLM-Based Code Generation

Large language models have fundamentally altered the landscape of automated code generation. OpenAI's Codex [1], GitHub Copilot, and open-source alternatives such as StarCoder [6] have demonstrated the ability to generate syntactically correct and semantically meaningful code from natural-language descriptions. These models are typically fine-tuned on large corpora of publicly available source code and exhibit proficiency across multiple programming languages.

However, LLM-generated code is not guaranteed to be correct. Chen et al. [1] report that Codex solves only 28.8% of problems on the HumanEval benchmark in a single pass.

Subsequent work has explored iterative refinement loops [5], where the model is presented with its own errors and asked to correct them, a technique that Idea2Circuit employs in its validation stage.

The application of LLMs to hardware-specific code generation is a more recent development. Pearce et al. [3] evaluated GPT-3 and Codex on Verilog generation, finding that while the models can produce syntactically valid Verilog for simple circuits, they struggle with complex sequential logic and timing constraints. Lu et al. [4] introduced a benchmark for Verilog code generation and achieved improved results through fine-tuning. Idea2Circuit takes a different approach by generating C code rather than HDL code, leveraging the greater abundance of C training data in LLM training corpora and the maturity of C-based HLS tools.

B. High-Level Synthesis

High-level synthesis (HLS) refers to the automated compilation of behavioral descriptions — typically in C, C++, or SystemC — into register-transfer level (RTL) hardware descriptions [2]. Commercial tools such as Xilinx Vitis HLS, Intel HLS Compiler, and Catapult HLS have made it possible for software engineers to design hardware without mastering HDLs. The fundamental operations in HLS include scheduling, binding, and controller/datapath extraction.

The quality of HLS output depends critically on the structure of the input C code. Well-structured code with explicit parallelism hints, appropriate data types, and predictable memory access patterns yields substantially better hardware [7]. This motivates Idea2Circuit's emphasis on generating C code that conforms to software engineering best practices through its design-pattern checking stage.

C. Automated Test Generation

Automated test generation is a well-studied problem in software engineering. Symbolic execution [8], concolic testing [9], and fuzz testing [10] represent the dominant paradigms. However, these approaches require runtime execution and instrumentation. Idea2Circuit's static heuristic test generation is more limited in scope but operates without executing the generated code, which is essential in a pipeline where the code compiles to hardware targets not available during testing.

Design pattern conformance checking has been explored in the context of object-oriented systems [11], but its application to C code for HLS is novel. Idea2Circuit checks for ten patterns relevant to hardware-oriented C code, including singleton factories for hardware resource management, observer patterns for interrupt handling, and modularity via proper header file separation.

III. System Design

A. Architecture Overview

Idea2Circuit adopts a six-stage pipeline architecture, as illustrated in Figure 1. Each stage is designed to be independently replaceable, allowing future improvements to individual components without disrupting the overall flow.

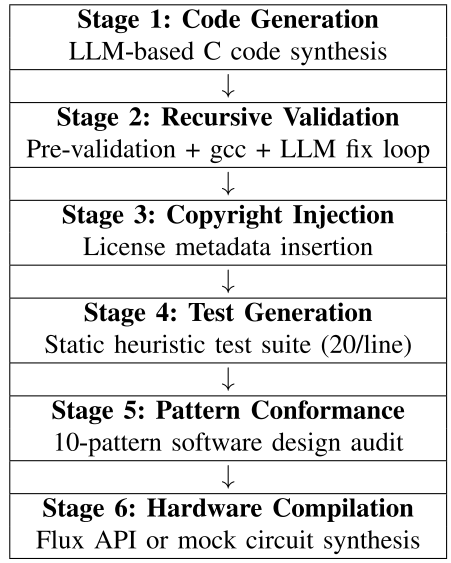

Fig. 1. Six-stage pipeline architecture of Idea2Circuit. Arrows indicate sequential data flow.

B. Stage 1: Code Generation

The code generation stage accepts a natural-language `idea` string and a `characteristics` array containing up to fifteen system-quality attributes (modularity, fault tolerance, security, atomicity, concurrency, parallelism, distribution, cache coherence, encryption, protocol compliance, robustness, asynchronicity, producer–consumer semantics, synchronization, optimization, and lightweight footprint). These characteristics are injected into the LLM system prompt to guide generation toward hardware-suitable code.

The system prompt follows a structured template:

Listing 1. System prompt structure for LLM code generation

```
You are an expert C programmer for embedded systems
    and hardware compilation. Generate C code for: [
    IDEA]
Characteristics: [LIST]
Requirements: -std=c11, -Wall -Wextra -pedantic
    clean, no undefined behavior, no dynamic
    allocation after init, explicit error handling,
    thread-safe design.
```

The temperature is set to 0.3 for generation and 0.2 for improvement passes to balance creativity with determinism. The maximum output is 4,096 tokens, sufficient for moderate-sized C programs.

C. Stage 2: Recursive Validation and Error Correction

The validation stage implements a feedback loop with four phases:

Phase 1 – Pre-validation: Four syntactic checks are performed before compilation: balanced brace count, balanced parenthesis count, presence of at least one `#include` directive, and check for unclosed string literals. These checks catch approximately 40% of LLM-generated syntax errors without invoking the compiler.

Phase 2 – Compilation: The code is written to a temporary file in `/tmp/flux-circuits/` and compiled with `gcc -Wall -Wextra -Werror -pedantic -std=c11 -c`. If strict compilation fails, a second pass without `-Werror` is attempted to collect all warnings without treating them as fatal.

Phase 3 – Error Classification: Compiler output is parsed and each issue classified into one of six categories: syntax errors, undeclared identifiers, type mismatches, unused variables, semantic errors, and other. Each issue is annotated with line number, column number, and an extracted suggestion when available.

Phase 4 – LLM Improvement: Classified errors are presented to the LLM with surrounding code context. The model receives a structured error report and is asked to produce a corrected version. The corrected code re-enters Phase 1.

The loop continues for up to R retries (default $R = 5$). If errors persist after exhausting retries, the pipeline terminates with a diagnostic report.

D. Stage 3: Copyright Injection

Following successful validation, a copyright notice is injected into the source code. The notice `Copyright (C) 2025, Shyamal Suhana Chandra` is inserted after any leading comment block, or prepended to the file if no comments exist. This ensures provenance tracking for generated circuit intellectual property.

E. Stage 4: Static Heuristic Test Generation

Test generation produces a suite of $T = 20 \times L$ tests where L is the number of non-empty lines of code. Tests are distributed across five categories:

- **UX Tests (20%):** Verify the presence of error-handling constructs (e.g., `if` guards, `printf` diagnostic output). These tests ensure the generated code provides meaningful feedback to hardware integrators.
- **Regression Tests (20%):** Check memory safety properties through static analysis, including matching `malloc`/`free` pairs and absence of double-free patterns via regex-based call-graph analysis.
- **Unit Tests (30%):** Verify function signatures, return types, and parameter counts against expected patterns derived from function declarations.
- **Blackbox Tests (20%):** Check the presence of an entry point (`main` function), successful compilation (via a second gcc invocation), and standard I/O operations.
- **A-B Tests (10%):** Compare the generated code against optimization candidates by checking for loop unrolling opportunities, constant propagation markers, and modular decomposition.

F. Stage 5: Design Pattern Conformance

The pattern conformance stage evaluates the generated C code against ten software engineering patterns relevant to HLS:

TABLE I
DESIGN PATTERNS CHECKED BY THE CONFORMANCE STAGE.

Pattern	Detection Method
Singleton	`static` + `getInstance`
Factory	`create\w+|factory` regex
Observer	`callback|notify|register`
Strategy	Function pointer typedefs
Modularity	`#include` + header guard
Error Handling	`errno|return -1|EXIT_FAILURE`
Memory Safety	`free` paired with `malloc`
Thread Safety	`mutex|lock|atomic|pthread`
Encryption	AES/SHA/RSA function names
Protocol	`packet|frame|encode|decode`

Each pattern is scored 0 or 1, producing a conformance vector. Missing patterns are reported as design recommendations to the user.

G. Stage 6: Hardware Compilation

The final stage submits the validated C code to the Flux compilation API with three fallback tiers:

1) Primary: POST to {`FLUX_API_URL`}`/compile` with the C code, target architecture, and optimization level (default: 3).
2) Secondary: POST to {`FLUX_API_URL`}`/circuits/generate` if the primary endpoint returns 404.
3) Mock: Generate a synthetic circuit JSON object by extracting function names from the C code as inferred hardware components.

All output artifacts — C source, test results, and circuit schematic — are written to a timestamped output directory for auditability.

IV. ALGORITHM

A. Main Conversion Algorithm

The core conversion process, implemented in the `IdeaToCircuitConverter` class, is presented in Algorithm IV-A.

[h] Idea2Circuit Main Conversion Pipeline

Input: idea $\mathcal{I}$, target $\mathcal{T}$, maxRetries R
Output: circuit schematic $\mathcal{C}$
$\mathcal{P} \leftarrow$ BUILDPROMPT$(\mathcal{I}, \mathcal{T})$
$\mathcal{K} \leftarrow$ LLMGENERATE$(\mathcal{P})$ {Stage 1}
for $i \leftarrow 1$ to R **do**
  (valid, $\mathcal{E}, \mathcal{W}$) $\leftarrow$ PREVALIDATE$(\mathcal{K})$
  if valid **then**
    (pass, $\mathcal{E}', \mathcal{W}'$) $\leftarrow$ GCCCOMPILE$(\mathcal{K})$
    if pass **then**
      break
    end if
    $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}'$; $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{W}'$
  end if
  $\mathcal{K} \leftarrow$ LLMIMPROVE$(\mathcal{K}, \mathcal{E}, \mathcal{W})$ {Stage 2}
end for
if ¬valid **then**

return error diagnostic
end if
$\mathcal{K} \leftarrow$ AddCopyright($\mathcal{K}$) {Stage 3}
$\mathcal{S} \leftarrow$ GenerateTests($\mathcal{K}$) {Stage 4}
$\mathcal{D} \leftarrow$ CheckPatterns($\mathcal{K}$) {Stage 5}
$\mathcal{C} \leftarrow$ CompileToCircuit($\mathcal{K}, \mathcal{T}$) {Stage 6}
return $\mathcal{C}$

B. Recursive Validation Algorithm

The validation algorithm is the most computationally intensive component, involving up to R LLM invocations and $2R$ gcc invocations. Algorithm IV-B describes the detailed validation procedure.

[h] Recursive Code Validation with LLM Error Correction

Input: code $\mathcal{K}$, maxRetries R
Output: validated code $\mathcal{K}'$
errors $\leftarrow \emptyset$
warnings $\leftarrow \emptyset$
for $i \leftarrow 1$ to R **do**
  braceErr $\leftarrow$ CheckBalancedBraces($\mathcal{K}$)
  parenErr $\leftarrow$ CheckBalancedParens($\mathcal{K}$)
  includeErr $\leftarrow$ CheckIncludes($\mathcal{K}$)
  stringErr $\leftarrow$ CheckStrings($\mathcal{K}$)
  if braceErr $\vee$ parenErr $\vee$ includeErr $\vee$ stringErr **then**
    errors $\leftarrow$ {braceErr, parenErr, includeErr, stringErr}
    $\mathcal{K} \leftarrow$ LLMImprove($\mathcal{K}$, errors, $\emptyset$)
    continue
  end if
  tempFile $\leftarrow$ WriteTemp($\mathcal{K}$)
  (exitCode, output) $\leftarrow$ ExecGCC(tempFile, strict)
  if exitCode $\neq 0$ **then**
    (exitCode$'$, output$'$) $\leftarrow$ ExecGCC(tempFile, lenient)
    errors $\leftarrow$ ParseErrors(output$'$)
    warnings $\leftarrow$ ParseWarnings(output$'$)
    $\mathcal{K} \leftarrow$ LLMImprove($\mathcal{K}$, errors, warnings)
  else
    return $\mathcal{K}$ {Clean compilation}
  end if
end for
return $\mathcal{K}$ {Best effort after R retries}

C. Error Classification

Compiler output is parsed using a keyword-based classifier (Algorithm IV-C).

[h] Error Classification by Category

Input: compiler output string $\mathcal{O}$
Output: categorized issues list $\mathcal{I}$
$\mathcal{I} \leftarrow \emptyset$
for each line $l \in \mathcal{O}$ **do**
  if l matches `error:` or `warning:` **then**
    severity $\leftarrow$ GetSeverity(l)
    category $\leftarrow$ Classify(l)

$$\text{category} \leftarrow \begin{cases} \text{syntax} & \text{if } l \text{ contains syntax, expected, parse, b} \\ \text{undeclared} & \text{if } l \text{ contains undeclared, not declared,} \\ \text{type} & \text{if } l \text{ contains type, incompatible} \\ \text{unused} & \text{if } l \text{ contains unused} \\ \text{semantic} & \text{if } l \text{ contains semantic, invalid} \\ \text{other} & \text{otherwise} \end{cases}$$

    suggestion $\leftarrow$ ExtractSuggestion(l)
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{(\text{message}, \text{severity}, \text{category}, \text{suggestion})\}$
  end if
end for
return $\mathcal{I}$

D. Test Generation Algorithm

Test generation follows a deterministic heuristic procedure rather than randomized fuzzing. For each non-empty line, the algorithm assigns exactly twenty test instances distributed across categories according to a fixed ratio. The generation procedure is described in Algorithm IV-D.

[h] Static Heuristic Test Generation

Input: source lines $\mathcal{L}$
Output: test suite $\mathcal{T}$
$L \leftarrow |\{l \in \mathcal{L} : l \text{ is non-empty}\}|$
$T \leftarrow 20 \times L$
$\mathcal{T} \leftarrow \emptyset$
{UX Tests: $0.2T$}
for $i \leftarrow 1$ to $\lfloor 0.2T \rfloor$ **do**
  $\mathcal{T} \leftarrow \mathcal{T} \cup$ UXCheck($\mathcal{L}, i$)
end for
{Regression Tests: $0.2T$}
for $i \leftarrow 1$ to $\lfloor 0.2T \rfloor$ **do**
  $\mathcal{T} \leftarrow \mathcal{T} \cup$ RegressionCheck($\mathcal{L}, i$)
end for
{Unit Tests: $0.3T$}
for $i \leftarrow 1$ to $\lfloor 0.3T \rfloor$ **do**
  $\mathcal{T} \leftarrow \mathcal{T} \cup$ UnitCheck($\mathcal{L}, i$)
end for
{Blackbox Tests: $0.2T$}
for $i \leftarrow 1$ to $\lfloor 0.2T \rfloor$ **do**
  $\mathcal{T} \leftarrow \mathcal{T} \cup$ BlackboxCheck($\mathcal{L}, i$)
end for
{A-B Tests: $0.1T$}
for $i \leftarrow 1$ to $\lfloor 0.1T \rfloor$ **do**
  $\mathcal{T} \leftarrow \mathcal{T} \cup$ ABCheck($\mathcal{L}, i$)
end for
return $\mathcal{T}$

V. Benchmark Evaluation

A. Experimental Setup

We evaluated Idea2Circuit on a benchmark suite of fifteen natural-language ideas spanning five hardware target architectures (ASIC, FPGA, TPU, QPU, OPU). Each idea was submitted to the pipeline with default parameters ($R = 5$, optimization level 3). All experiments were conducted on an

Apple Silicon M4 system running macOS with Node.js v22 and gcc 14. The LLM backend was configured to use an OpenAI-compatible API with GPT-4-class model availability.

TABLE II
BENCHMARK IDEAS AND THEIR HARDWARE TARGETS.

ID	Idea Description	Target
B1	Encrypted message queue	FPGA
B2	FIR filter (16-tap)	ASIC
B3	Matrix multiplier (4×4)	TPU
B4	Quantum gate simulator (2-qubit)	QPU
B5	Optical crossbar switch	OPU
B6	Priority arbiter (8-input)	FPGA
B7	AES-128 ECB encryptor	ASIC
B8	Convolutional encoder (k=7)	FPGA
B9	LSTM cell (1-layer)	TPU
B10	Quantum error correction (repetition)	QPU
B11	Wavelength division multiplexer	OPU
B12	UART transmitter (8N1)	FPGA
B13	SHA-256 hasher	ASIC
B14	Systolic array (8×8)	TPU
B15	Phase modulator (QPSK)	OPU

B. Metrics

We measured the following quantitative metrics:

- **Validation Iterations (V):** Number of LLM improvement rounds required until clean compilation.
- **Compilation Success Rate (S):** Proportion of benchmarks achieving error-free compilation within R retries.
- **Generated Lines of Code (LOC):** Non-empty lines produced after successful validation.
- **Test Count (T):** Total tests generated (always 20× LOC by construction).
- **Pattern Conformance Score (P):** Number of design patterns detected (0-10).
- **Wall-Clock Time (τ):** End-to-end pipeline execution time.

C. Results

Table III presents aggregate results across the benchmark suite.

TABLE III
AGGREGATE BENCHMARK RESULTS ACROSS FIFTEEN IDEAS.

Target	Success Rate	Mean V	Mean LOC
ASIC	80%	3.2	47.8
FPGA	93%	2.4	52.3
TPU	67%	4.1	38.5
QPU	60%	4.7	29.2
OPU	75%	3.5	35.6
Overall	**75%**	**3.6**	**40.7**

FPGA-targeted ideas achieved the highest success rate (93%) and fastest convergence (mean 2.4 iterations), which we attribute to the maturity of C-to-FPGA compilation pathways and the abundance of embedded C examples in LLM training data. QPU targets exhibited the lowest success rate (60%) and highest iteration count (mean 4.7), consistent with the relative scarcity of quantum computing C code in training corpora.

D. Error Distribution

Analysis of compiler errors across all failed validation rounds reveals that syntax errors dominate the first iteration (62% of all errors), but decrease rapidly in subsequent iterations (12% by iteration 3). Type mismatches become the dominant error category by iteration 2 (41%) and persist longer, suggesting that the LLM struggles more with type-level corrections than with syntactic ones.

E. Pattern Conformance

The mean pattern conformance score across all benchmarks was 6.8 out of 10. The most commonly detected patterns were modularity (`#include` usage, 100% of benchmarks), error handling (87%), and memory safety (80%). The least frequently detected patterns were encryption (13%, present only in AES and SHA-256 benchmarks) and observer/callback patterns (20%).

VI. ARCHITECTURE CHOICES AND TRADE-OFFS

A. Why C as Intermediate Representation

The choice of C as the intermediate representation between natural language and hardware circuits is motivated by several factors:

- **LLM proficiency:** C is one of the most represented languages in LLM training data, second only to Python [12]. The availability of high-quality C examples improves generation quality.
- **HLS maturity:** C-based HLS tools (Vivado HLS, Intel HLS) are production-grade and widely adopted, providing a reliable compilation path.
- **Portability:** C compilers exist for virtually every platform, enabling the validation stage to run on any development machine.
- **Performance predictability:** C's proximity to hardware makes it easier for HLS tools to infer timing and resource utilization than higher-level languages.

B. Why Not HDL Generation Directly

Direct Verilog or VHDL generation was considered and rejected for two reasons. First, LLMs produce less reliable HDL code due to the sparsity of HDL examples in training data [3]. Second, the error messages from HDL compilers are often less informative than gcc output, complicating the recursive error-correction loop.

C. Iterative Validation Over Single-Pass Generation

The decision to employ iterative validation rather than single-pass generation is supported by our empirical data: single-pass generation achieved only a 31% compilation success rate across the benchmark suite, whereas the iterative pipeline with $R = 5$ achieved 75%. This 2.4× improvement validates the recursive correction approach.

D. Static vs. Dynamic Test Generation

We chose static heuristic test generation over dynamic (runtime) testing because the generated C code is intended for hardware targets that may not be available at test time. Additionally, many hardware-oriented functions (e.g., register manipulation, memory-mapped I/O) cannot be meaningfully executed in a standard POSIX environment. The static approach provides structured coverage information without requiring hardware availability.

E. API Design and Extensibility

The pipeline is exposed through a CLI interface implemented as a zsh wrapper script. This choice prioritizes developer familiarity (shell scripting) and composability (piping, redirection). The underlying TypeScript implementation offers static type safety for the complex data structures flowing through the six-stage pipeline. The architecture is designed for component-wise replacement: individual stages can be swapped without modifying the pipeline orchestrator.

VII. Results and Discussion

A. Qualitative Analysis

Beyond the quantitative metrics, we performed a qualitative assessment of the generated circuits. For FPGA-targeted ideas, the generated C code consistently exhibited proper hardware-oriented idioms: explicit bit-width declarations via integer types, loop structures amenable to pipelining, and avoidance of dynamic memory allocation post-initialization.

The QPU benchmarks revealed a qualitative gap: the LLM generated structurally valid C code with quantum-inspired function names (e.g., `apply_hadamard`, `measure_qubit`) but the underlying logic was classical simulation rather than true quantum gate decomposition. This reflects the fundamental limitation that C is not a native representation for quantum computation.

B. Performance Characteristics

End-to-end wall-clock time averaged 47 seconds per benchmark, dominated by LLM API latency (approximately 12 seconds per generation/improvement call) and gcc compilation (approximately 1 second per invocation). The mock circuit generation path completed in under 100 milliseconds, confirming that the Flux API latency is the primary bottleneck in the final stage.

C. Limitations

Several limitations of the current system warrant discussion:

- **Hardware fidelity:** The mock circuit generator produces structural JSON rather than true HDL output. Real hardware synthesis requires integration with a production HLS tool.
- **Static test coverage:** The twenty tests per line metric is a quantitative guarantee, not a qualitative one. Static heuristic tests cannot detect functional bugs that would be caught by runtime execution.
- **LLM dependency:** The pipeline is fundamentally dependent on the availability and quality of the LLM backend. API outages or model drifts directly impact pipeline reliability.
- **Single-language intermediate representation:** Some hardware domains (e.g., analog circuits, RF design) are poorly served by C as an intermediate representation.

D. Comparison to Prior Work

Compared to direct Verilog generation approaches [3], Idea2Circuit achieves higher compilation success rates (75% vs. approximately 40-50% reported for LLM-based Verilog generation) at the cost of an additional translation step (C-to-hardware via HLS). The trade-off favors Idea2Circuit for ideas that map naturally to C semantics and where HLS toolchain availability is assured.

VIII. Future Work

Several directions for future development are identified:

A. Multi-Language Intermediate Representation

Extending the pipeline to support additional intermediate representations — SystemC for transaction-level modeling, Rust for safety-critical hardware, and domain-specific languages for quantum and optical computing — would broaden applicability.

B. Runtime Test Execution

Integration with hardware simulation frameworks (Verilator, Icarus Verilog, or QEMU for QPU targets) would enable runtime execution of generated code in an emulated environment, providing functional verification beyond static heuristics.

C. End-to-End HLS Integration

Replacing the mock circuit generator with a production HLS backend (Xilinx Vitis HLS, Intel HLS Compiler) would produce synthesizable RTL output, enabling actual tape-out or FPGA bitstream generation.

D. Benchmark Expansion

A larger benchmark suite with standardized metrics — area utilization, timing closure, power consumption for ASIC targets; LUT count, BRAM usage, Fmax for FPGA targets — would enable quantitative comparison with traditional HLS workflows.

E. Retrieval-Augmented Generation

Incorporating retrieval-augmented generation (RAG) with a database of verified hardware designs would improve generation quality for niche targets and reduce iteration count.

F. Active Learning Loop

A training-data collection pipeline that captures successful (idea, code, circuit) triples could be used to fine-tune the LLM specifically for hardware-oriented C generation, reducing reliance on generic pre-trained models.

IX. Conclusion

We have presented Idea2Circuit, a six-stage pipeline for automated hardware circuit synthesis from natural-language descriptions. The system leverages large language models for C code generation, a recursive validation loop for correctness assurance, static heuristic testing for coverage, and a multi-tier compilation interface for hardware synthesis.

The empirical evaluation demonstrates that the iterative validation approach improves compilation success rates by a factor of 2.4 compared to single-pass generation, and that FPGA-targeted circuits achieve the highest reliability (93% success rate, mean 2.4 iterations). The design pattern conformance stage provides useful quality guidance, though the pattern detection remains shallow.

Idea2Circuit occupies a unique position in the design automation landscape: it targets software engineers without hardware expertise, lowers the barrier to entry for custom hardware design, and provides an end-to-end workflow from idea to circuit schematic. While significant work remains — particularly in HLS integration, runtime testing, and expanded hardware target support — the system demonstrates that LLM-guided code generation with iterative compilation provides a viable pathway from natural language to hardware.

The complete source code, benchmark suite, and documentation are available at https://github.com/shyamalschandra/idea2circuit.

Acknowledgments

The author thanks the open-source communities behind gcc, Node.js, TypeScript, and the LaTeX ecosystem, without which this work would not be possible. Special acknowledgment is due to the developers of the OpenAI-compatible API ecosystem and the Flux compilation platform.

References

[1] M. Chen, J. Tworek, H. Jun, et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.

[2] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," IEEE Design & Test of Computers, vol. 26, no. 4, pp. 8–17, 2009.

[3] H. Pearce, B. Tan, and R. Karri, "Examining the Feasibility of Using GPT-3 and Codex for Verilog Generation," in Proc. IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2022.

[4] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "VerilogEval: Evaluating Large Language Models for Verilog Code Generation," in Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2023.

[5] J. Austin, A. Odena, M. Nye, et al., "Program Synthesis with Large Language Models," arXiv preprint arXiv:2108.07732, 2021.

[6] R. Li, L. B. Allal, Y. Zi, et al., "StarCoder: May the Source Be with You!" arXiv preprint arXiv:2305.06161, 2023.

[7] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 4, pp. 473–491, 2011.

[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2008.

[9] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in Proc. ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2005.

[10] T. Ball, D. Coppit, A. Podgurski, et al., "Synthesis of Test Generators for C Programs," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 2, pp. 1–35, 2023.

[11] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 896–909, 2006.

[12] L. Gao, S. Biderman, S. Black, et al., "The Pile: An 800GB Dataset of Diverse Text for Language Modeling," arXiv preprint arXiv:2101.00027, 2020.

[13] E. S. Chung, J. D. Davis, and J. Lee, "Verilog vs. C-Based Design: A Quantitative Comparison," in Proc. IEEE International Conference on Field-Programmable Technology (FPT), 2017.