PICBench: Benchmarking LLMs for Photonic Integrated Circuits Design

Yuchao Wu HKUST(GZ) | Xiaofei Yu HKUST(GZ) | Hao Chen HKUST(GZ) | Yang Luo HKUST(GZ) | Yeyu Tong HKUST(GZ) | Yuzhe Ma HKUST(GZ)

***Abstract*—While large language models (LLMs) have shown remarkable potential in automating various tasks in digital chip design, the field of Photonic Integrated Circuits (PICs)—a promising solution to advanced chip designs—remains relatively unexplored in this context. The design of PICs is time-consuming and prone to errors due to the extensive and repetitive nature of code involved in photonic chip design. In this paper, we introduce PICBench, the first benchmarking and evaluation framework specifically designed to automate PIC design generation using LLMs, where the generated output takes the form of a netlist. Our benchmark consists of dozens of meticulously crafted PIC design problems, spanning from fundamental device designs to more complex circuit-level designs. It automatically evaluates both the syntax and functionality of generated PIC designs by comparing simulation outputs with expert-written solutions, leveraging an open-source simulator. We evaluate a range of existing LLMs, while also conducting comparative tests on various prompt engineering techniques to enhance LLM performance in automated PIC design. The results reveal the challenges and potential of LLMs in the PIC design domain, offering insights into the key areas that require further research and development to optimize automation in this field. Our benchmark and evaluation code is available at https://github.com/PICDA/PICBench.**

I. Introduction

Photonic Integrated Circuits (PICs) represent a groundbreaking advancement in chip design, harnessing the properties of light to enable faster data processing and greater energy efficiency. As the demand for high-performance computing and communication systems continues to rise, PICs have emerged as a critical solution to meet these requirements [1]–[4]. However, unlike the maturity of electronic design automation (EDA) tools, the development of photonic design automation (PDA) tools capable of supporting automated design pipelines for circuit simulation and layout remains at an early stage. The design and layout of photonic circuits and components still heavily rely on manual input, which introduces significant inefficiencies. Photonic designs are inherently complex, often requiring repetitive, low-level coding for devices and connections. This process is time-intensive and prone to human error, particularly as the size and complexity of the designs increase. Consequently, there is an urgent need for a comprehensive set of tools to fully automate photonic circuit design and layout processes. Advancements in large language models (LLMs) [5]–[7] offer a promising opportunity to address these challenges and accelerate the development of PDA solutions.

Recently, LLMs have demonstrated significant potential in automating code generation for hardware designs, which offers substantial support to engineers in designing and verifying these systems. RTLLM [8] introduced a benchmark framework comprising 30 designs spanning diverse complexities and scales for Verilog generation. Then VerilogEval [9] proposed an extensive dataset of 156 problems and a robust testing procedure to facilitate the systematic evaluation of generated code. Beyond Verilog generation, SPICEPilot [10] investigated the capabilities of LLMs in generating SPICE code. Another work ChatEDA [11] demonstrated the ability to generate code for interacting with EDA tools using natural language instructions.

Nevertheless, the application of LLM in photonic circuit design has been limited to a few works. Li *et al*. [12] utilized LLM generating FDTD code for simulating the photonic crystal surface emitting laser (PCSEL) structure and AI code for subsequent optimizations of the PCSEL model. However, their approach was not fully automated, as it relied heavily on human experts to iteratively specify requirements and debug errors. Liu *et al*. [13] presented an automated framework that translates natural language prompts into Python code capable of generating GDSII files using an open-source library. However, their framework was tested on only seven simple device designs, leaving the performance and scalability of LLM-based solutions insufficiently evaluated. The absence of a reliable and automated testing framework, coupled with limited datasets and the lack of a standardized benchmark, significantly hinders both the development and fair evaluation of LLM solutions in PICs design. To address these challenges, a comprehensive benchmark is needed—one that encompasses a wide range of design problems, includes a reliable and automated evaluation framework to minimize testing variance, and clearly distinguishes the correctness and efficiency of solutions.

In this paper, we introduce PICBench, an open-source and comprehensive benchmark for PIC design using natural language to generate simulation-ready netlists. The benchmark includes 24 meticulously crafted PIC design problems, covering a wide range of design complexities and scales. Each problem features clear descriptions and is accompanied by ground-truth designs created by human experts, serving as a golden result for evaluation. Leveraging an open-source simulator SAX [14], PICBench enables efficient and automated evaluation of any LLM-generated results.

Our contributions are summarized as follows:

- We introduce PICBench, the first comprehensive open-source benchmark for PIC design using LLMs, comprising 24 carefully designed PIC design problems.
- We proposed a simple but efficient feedback-based method that further enhances the model's proficiency in PIC design tasks.

978-3-9826741-0-0/DATE25/© 2025 EDAA

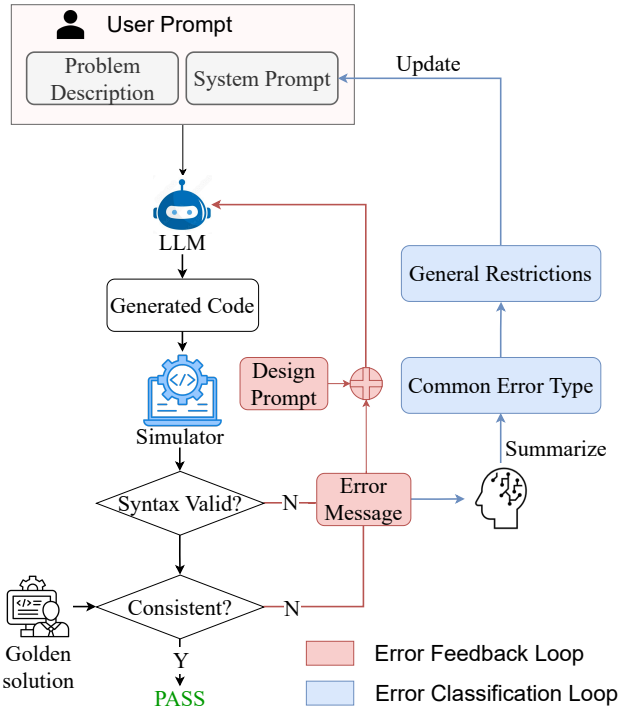

Fig. 1 PICBench framework that automated design generation and evaluation.

- We exhaustively evaluated the state-of-the-art commercial LLMs with our benchmark on both syntax and functionality.

II. Preliminary

In this section, we will first introduce the open-source PIC simulator SAX [14] and then present our description of the PIC design task based on natural language instructions.

A. SAX

SAX is a Python library designed for S-parameter-based circuit simulation and optimization in the frequency domain, leveraging JAX for automatic differentiation and GPU acceleration. It provides a functional approach to modeling photonic integrated circuits, allowing users to define components and circuits using standard Python functions and dictionaries. Given a JSON netlist specifying input/output ports, required components, their configurations, and detailed interconnections, SAX can efficiently perform mathematical analysis and simulate circuit behavior.

B. Task Description

Here we describe the PIC design task based on natural language instructions as follows:

- Given the natural language description of the desired circuit functionality and specified configurations, the objective is to understand and respond to user requirements and generate the simulate-ready netlist of this design.

III. PICBench

In this section, we will first present an overview of our PICBench framework and then introduce the details.

A. Framework

Fig. 1 illustrates PICBench's flow: user provides a natural language description of their PIC design task to the LLM. The

Problem Description

Create a Mach-Zehnder interferometer (MZI) with a single input and output, featuring a path length difference of ΔL. A phase shifters with a length of L should be applied to the top arm to modulate the phase of the optical signal. Use the built-in multimode interferometer (MMI) component for splitting and combining the optical signals, and the built-in phase shifters to achieve the desired phase modulation.

Parameters:
ΔL = 10 microns;
L = 10 microns

Fig. 2 Example of problem description.

LLM's output, a simulation-ready netlist, is then directly fed into SAX for simulation. If the simulation tool reports errors, the tool's outputs are returned to the LLM as a new prompt with a request to rectify the errors. Simultaneously, the error information is reviewed and summarized into restrictions by human inspection, which are then incorporated into the initial system prompt. If the simulation tool successfully generates a frequency response, PICBench tests the functionality of the generated design by comparing it with the golden results. The process terminates when both syntax and functionality tests are passed. Otherwise, the process iterates up to a user-specified number of trials.

B. Detail Description of Our Benchmark

PICBench collects 24 meticulously crafted, commonly encountered PIC design problems, spanning a wide range of design complexities and scales. TABLE I shows the detailed description of all 24 problems provided in our benchmark, including 6 optical computing circuits, 7 optical interconnects circuits, 9 optical switches, and 2 fundamental devices. These problems exhibit significant variation in their target functionalities, including examples such as optical switches, optical modulators, and more. In addition to the diverse functionalities, our benchmark also exhibits rich variation in the implementation. For instance, optical switches in our benchmark share the common functionality of switching signal connections. However, we include all widely used architectures of optical switches, such as crossbar, Spanke, Benes, and Spanke-Benes, with configurations ranging from 4×4 to 8×8. Notably, we do not include any purely device-level design problems in our collection, as these lack connections, and the components section in each netlist inherently addresses device-level designs. The foundational devices we include are not simple device components. They involve connections among more than two components and can serve as the basis for constructing more complex circuits.

For each design, we provide three key components: a detailed problem description, the correct design, and its corresponding frequency response. The problem description is a natural language description of the desired circuit functionality, including the required configurations and the number of input and output ports, as illustrated in Fig. 2, which provides an example of the MZI_ps problem. Human designers then manually craft correct design based on the description, producing golden solution for evaluation. To streamline the evaluation process, the correct design is subsequently fed into the simulator, and its frequency response is directly saved. This frequency response serves as a

TABLE I Benchmark Description

Design		Description
Optical Computing	Clements_4 × 4	A 4 × 4 MZI mesh arranged using the Clements method
	Clements_8 × 8	An 8 × 8 MZI mesh arranged using the Clements method
	Reck_4 × 4	A 4 × 4 MZI mesh arranged using the Reck method
	Reck_8 × 8	An 8 × 8 MZI mesh arranged using the Reck method
	NLS	A Non-Linear Sign gate with a signal channel and two additional ancilla channels
	U-matrix block	A fundamental block representing a 2 × 2 unitary matrix of arbitrary values
Optical Interconnects	Direct modulator	An optical direct modulator
	QPSK modulator	An optical QPSK modulator
	8-QAM modulator	An optical 8-QAM modulator
	64-QAM modulator	An optical 64-QAM modulator
	WDM_mux	A WDM multiplexer
	WDM_demux	A WDM demultiplexer
	Optical hybrid	A 90° optical hybrid
Optical Switch	OS_2 × 2	A fundamental 2 × 2 optical switch
	Crossbar_4 × 4	A 4 × 4 optical switching network based on Crossbar architecture
	Crossbar_8 × 8	An 8 × 8 optical switching network based on Crossbar architecture
	Spanke_4 × 4	A 4 × 4 optical switching network based on Spanke architecture
	Spanke_8 × 8	An 8 × 8 optical switching network based on Spanke architecture
	Benes_4 × 4	A 4 × 4 optical switching network based on Benes architecture
	Benes_8 × 8	An 8 × 8 optical switching network based on Benes architecture
	Spanke–Benes_4 × 4	A 4 × 4 optical switching network based on Spanke–Benes architecture
	Spanke–Benes_8 × 8	An 8 × 8 optical switching network based on Spanke–Benes architecture
Fundamental Devices	MZM	A Mach-Zehnder modulator
	MZI_ps	A Mach-Zehnder interferometer with a phase shifter

reference for verifying the correctness of the design's functionality.

C. Code Generation and Evaluation

Since the netlist required by SAX does not follow a general format, we designed a system prompt template to maximize the efficiency of instructing LLMs to generate high-quality, error-free designs. As shown in Fig. 3, the template consists of three components:

1) **Required format**: This part provides the schema for the required format. It defines the structure of the netlist to ensure compliance with SAX's requirements.
2) **API document**: This part includes detailed documentation of the built-in components provided by SAX or defined by us. It specifies port definitions and configurable parameters, offering a comprehensive reference for implementing various components within the netlist.
3) **Restrictions**: This part establishes a set of clear restrictions designed to standardize and streamline the generation of netlist JSON content.

This structured approach promotes uniformity across all generated netlists, minimizes errors and ambiguity, and ensures that the outputs are both precise and efficient.

Once the design is successfully generated, its evaluation is automatically conducted, focusing primarily on two aspects: syntax and functionality.

Syntax correctness is the most fundamental requirement for ensuring logical functionality and executable code, as functionality cannot be assessed without first confirming syntactic validity. To verify syntax, the design is tested using the SAX simulator. If no errors are detected and a frequency response is successfully generated, the syntax is considered valid.

Functionality correctness is also evaluated through simulation to determine whether the generated design performs as expected. However, unlike traditional testbenches used in RTL design [8], where specific input signals are crafted to verify the correctness of outputs, our simulations are conducted in the frequency domain. In this context, the input corresponds to a range of frequencies rather than discrete signals, and the response at any single wavelength alone does not provide a conclusive indication of success or failure, nor does it enable precise and efficient feedback. Therefore, we simply compare the simulation results between generated code completions and golden reference solutions. Since the built-in components are limited, we manually construct all required components based on the descriptions provided in the API document, ensuring that all problems in our benchmark are successfully evaluated.

D. Error Classification Loop

The PIC netlist design generation problem involves a specialized language and task that is rarely encountered during the pre-training of existing LLMs, leading to inherent limitations in

System Prompt

You are a professional Photonic Integrated Circuit (PIC) designer. Your task is to generate a JSON netlist based on the user's design requirements. This netlist should specify input/output ports, the necessary components, their configurations, and detailed connections between them. You only complete chats with syntax correct JSON code and the format is as follows:

```
<<<JSON format>>>
{
 "netlist":{
  "instances": {
   "<component_name1>": "<component>",
   "<component_name2": {'component': '<component>', 'settings': {'<parameter>': <value>}}
   ...
  },
  "connections": {
   "<component_name>,<port>": "<component_name>,<port>",
   ...
  },
  "ports": {
   "<port_name>": "<component_name>,<port>",
   ...
  }
},
 "models":{
  "<component>": "<ref>",
     ...
 }
}
```

You have access to the following built-in devices, only these devices are permitted unless otherwise specified:

```
<<<API document>>>
mzi:
   description: Mach-Zehnder interferometer with one input and one output
   input ports: I1          output ports: O1
   parameters: delta length..
...
```

Note that:
1. Your answers should be professional and logical.
2. The analyses should be as detailed as possible. For example, you can think it step by step.
3. The response must consist of two sections:
  - analysis: A detailed explanation of how the netlist was generated. Start by <analysis>.
  - result: The generated netlist JSON content. Start by <result>. Only the JSON content is required in the result.
4. Never specify extra parameters unless explicitly stated in the instructions; always use default values. If a difference between two parameters is specified, use the default value for one and adjust the other by the specified difference.
5. The default unit is micron.
6. Unless otherwise specified, use built-in components to implement whenever possible. Never specify extra parameters if the instruction do not specify, always use the default value.

Fig. 3 System prompt template for code generation.

their performance on such tasks. Despite employing in-context learning and providing several examples, LLMs often confuse the PIC netlist format with other netlist formats. For instance, in a PIC netlist, each port can only be connected once, and duplicate connections to the same port are prohibited. However, LLMs frequently generate connections containing multi-pin nets, as seen in traditional VLSI netlists, which is incorrect.

To effectively employ LLMs in PIC design generation, we employ an automatic error classification feedback method. Since the specific aspects causing LLM failures are uncertain, errors are iteratively inspected and summarized during the generation process. For each conversation, if a syntax error is detected, a human expert inspects the error information, identifies common errors, and summarizes them into general restrictions to prevent the recurrence of similar errors. TABLE II summarize all the common error types we collected during our trials and the corresponding restrictions that we collect as prompt. These domain-specified restrictions are then integrated into system prompt to improve understanding of PIC modules and coding styles, and provide valuable insights into the primary reasons for failures. This prompt-tuning approach effectively addresses poor code generation, significantly enhancing LLM performance for PIC design tasks.

TABLE II Restrictions for the PIC design task, listing the main failure types and corresponding constraints to maintain valid syntax.

Failure Types	**Restrictions**
Use undefined models	Only built-in devices are permitted unless otherwise specified; never use undefined models.
Bind the I/O ports	Input or output ports in the `ports` section represent only the system's start or end points; they must not appear in any internal connections.
Mess up 'Instances' and 'models' part	When specifying built-in components, the model reference must appear in the `models` section like ``...: "<ref>"'' rather than ``"<ref>" : ...''. The `instances` section only instantiates these components.
Extra contents found in JSON	Only the required JSON netlist elements should appear in the output. Do not include comments, advice, or code block markings.
Duplicate connections to the same port	Each port can only be connected once; duplicate connections to the same port are prohibited.
Wrong connections for dangling ports	If a specific port mapping is not explicitly required, omit it rather than introducing arbitrary or unused port names.
Wrong ports number	The total number of input and output ports must align with the design specification. Each input port typically starts with `I`, and each output port with `O`.
Wrong ports	Ensure all `connections` and `ports` are valid and consistent with the defined `instances` and `models`. Do not generate invalid or undefined mappings.
Wrong component name	Underscores are prohibited in component names.
Other syntax error	\

E. Error Feedback Loop

Due to the inherent hallucination tendencies of LLMs, even when comprehensive restrictions are provided to mitigate trivial errors, the same issues may persist. Inspired by real-world coding practices—where code is rarely correct on the first attempt, and iterative feedback from simulation and synthesis tools is critical for meeting design specifications—we adopt a feedback-based method that efficiently leverages error information from each query and the simulator.

For each problem, the initial query follows the standard process introduced in Section III-C. However, if the simulator detects a syntax error, our correction feedback loop is triggered. The error is first classified into specific categories, as outlined in Section III-D, enabling the precise identification of its cause without requiring the LLM to interpret abstract error messages which can directly inform code refinement. Next, the error category, along with detailed error reports and a crafted feedback prompt, is fed back to the LLM to refine the previously generated code in a manner similar to human debugging. If the evaluation instead identifies a functional error, the feedback loop provides a concise prompt "The syntax is correct, but a functional error

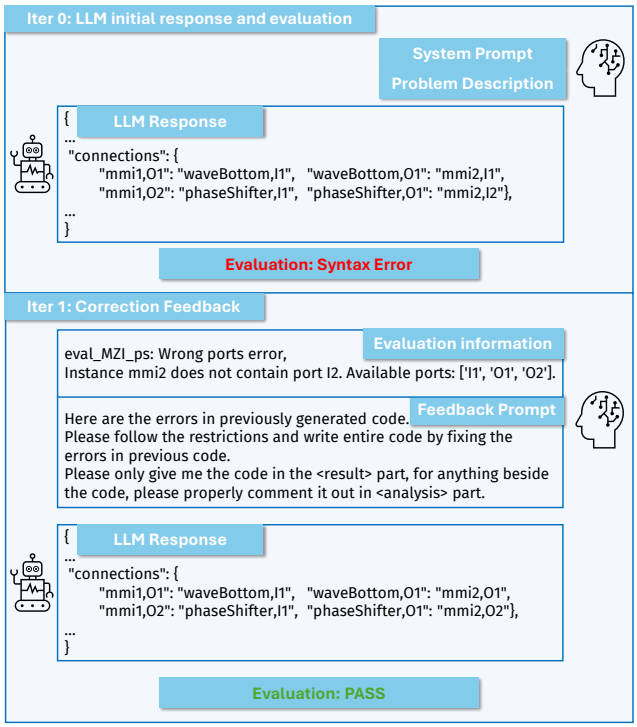

Fig. 4 An example of solving MZI_ps by GPT o1-mini with feedback

has occurred. Please review the problem description carefully". The feedback loop will continue iteratively until the code passes or the maximum number of iterations is reached.

Fig. 4 gives an example of MZI_ps to illustrate this process, Initially, the LLM generates a result that incorrectly connects to non-existent port I2. This error is automatically classified as a "Wrong Ports Error." The classification, combined with detailed error information and a crafted feedback prompt, is then fed back to the LLM. After one iteration of the correction feedback loop, the error is resolved, and the code passes the test.

IV. Experimental Results

A. Experiment Setup

We implemented the PICBench in Python. The quality of the given PIC design was evaluated using the open-source simulation tool SAX, which specializes in S-parameter-based circuit simulations. We constructed the S-parameters for essential devices, including waveguides, couplers, MMIs, MZIs, MRRs and phase shifters, to simulate the frequency-domain response of the specified PIC over the wavelength range of 1510 to 1590 nm.

PICBench is compatible with a wide range of LLMs as long as they provide a Python API. In our experiment, we extensively evaluated the capabilities of five notable LLMs developed by leading companies, using PICBench:

- **GPT-4**: The free commerical model developed by OPENAI.
- **GPT-4o**: The flagship commercial model that is widely recognized for its versatility and high accuracy across various domains.
- **GPT-o1-mini**: A smaller GPT-based model optimized for STEM reasoning, with a focus on mathematical and technical problem-solving.
- **Gemini 1.5 Pro**: Developed by Google, an emerging model that integrates hybrid architectures to improve code generation and context handling.
- **Claude 3.5 Sonnet**: Developed by Anthropic, emphasizing significant improvement in graduate-level reasoning, knowledge acquisition, and coding abilities.

We adopted the widely used Pass@k metric [15] to measure code generation correctness. A problem is considered solved if any of the k generated samples passes the corresponding unit tests. For each task, n samples (default $n = 5$) are generated, of which c samples pass, and an unbiased estimator of Pass@k is computed as:

$$\text{pass@k} := \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]. \quad (1)$$

To analyze the impact of error feedback (EF), we queried the selected LLMs both without and with feedback for n iterations where we set $n = 1$ and $n = 3$.

B. Results

1) Impact of error feedback

TABLE III presents the comprehensive evaluation results for both syntax and functionality across all five selected LLMs, using 0, 1, and 3 error feedback iterations with PICBench, evaluated through the Pass@k metric.

Without correction feedback, GPT-4 demonstrates the highest syntax accuracy, with 16.67% for Pass@1 and 41.67% for Pass@5, establishing its strength in pattern recognition and abstraction capabilities.

When error feedback is incorporated, there is a clear improvement in both syntax and functionality scores across Pass@1 and Pass@5 evaluations. Claude 3.5 Sonnet demonstrated the most significant improvement in both syntax performance and functionality in response to feedback, highlighting its excellent self-correction and adaptive learning capabilities. At Pass@1, its syntax score increased from 13.33% without feedback to 35.83% with one feedback iteration, and further soared to 75.83% with three iterations. Similarly, its functionality success rate improved from 1.67% to 24.17%.

Moreover, even one round of feedback can raise Pass@1 metrics beyond the Pass@5 results obtained without any feedback. This further emphasizes the impact of feedback on model performance. For example, Gemini 1.5 pro achieves a syntax score of 33.33% with one feedback iteration at Pass@1, which is notably higher than its Pass@5 syntax score without feedback (16.67%).

Overall, the feedback method, aided by simulator diagnostics, rapidly accelerates debugging and enables iterative improvements in code generation. By revealing each model's evolving logic and capacity for self-refinement, this approach delivers notable gains in both code quality and reliability.

2) Impact of restrictions

To investigate the impact of the restrictions introduced in Section III-D, we queried the selected LLMs using the same approach described in Section IV-B1.

As shown in the TABLE IV, the application of restrictions greatly enhances the performance of LLMs across all evaluated

TABLE III The Syntax and Functionality (Func.) evaluation for different LLMs. EF denotes the error feedback.

LLM	Pass@1						Pass@5					
	Without EF		With 1 EFs		With 3 EFs		Without EF		With 1 EFs		With 3 EFs	
	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.
GPT-4	**16.67**	6.67	34.17	6.67	54.17	10.83	**41.67**	12.50	**70.83**	16.67	**100.00**	29.17
GPT-o1-mini	8.33	4.17	33.33	15.00	63.33	23.33	29.17	**16.67**	66.67	**25.00**	91.67	33.33
GPT-4o	14.17	4.17	**40.83**	15.00	59.17	20.00	37.50	4.17	**70.83**	**25.00**	87.50	**41.67**
Claude 3.5 Sonnet	13.33	1.67	35.83	14.17	**75.83**	**24.17**	20.83	8.33	**70.83**	20.83	**100.00**	37.50
Gemini 1.5 pro	9.17	**8.33**	33.33	**16.67**	50.00	20.83	16.67	12.50	66.67	20.83	87.50	33.33

TABLE IV The Syntax and Functionality (Func.) correctness evaluation for different LLMs with our proposed restrictions. EF denotes the error feedback.

LLM	Pass@1						Pass@5					
	Without EF		With 1 EFs		With 3 EFs		Without EF		With 1 EFs		With 3 EFs	
	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.
GPT-4 + restrictions	20.00	4.17	38.33	17.50	71.67	30.00	58.33	12.50	58.33	20.83	**100.00**	45.83
GPT-o1-mini + restrictions	13.33	9.17	50.00	22.50	84.17	33.33	25.00	12.50	79.17	33.33	**100.00**	50.00
GPT-4o + restrictions	60.83	20.00	86.67	31.67	**95.00**	36.67	87.50	**37.50**	**100.00**	37.50	**100.00**	50.00
Claude 3.5 Sonnet + restrictions	54.17	15.83	80.83	29.00	90.00	**43.33**	87.50	**37.50**	**100.00**	**45.83**	**100.00**	**62.50**
Gemini 1.5 pro + restrictions	**64.17**	**21.67**	**88.33**	**32.50**	**95.00**	38.33	**91.67**	**37.50**	**100.00**	**45.83**	**100.00**	54.17

conditions, with notable improvements in both syntax and functionality. While all models benefit, the degree of improvement varies, with Gemini 1.5 pro showing the most dramatic enhancements across both syntax and functionality dimensions. For instance, Gemini 1.5 pro's syntax score improves from 9.17% to 64.17% in Pass@1 and from 16.67% to 91.67% in Pass@5, while its functionality score increases from 8.33% to 21.67% in Pass@1 and from 12.50% to 37.50% in Pass@5 with restrictions applied, demonstrating its robust in-context reasoning and real-time adaptability. The results highlight the potential for further optimization by leveraging the in-context learning ability of LLMs, enhanced with high-definition circuit knowledge.

V. Conclusion

In this paper, we propose a comprehensive open-source benchmark for PIC design generation using LLMs, featuring 24 meticulously crafted PIC design problems. We also introduce a feedback-based prompt engineering technique that iteratively refines designs and enhances the models' design generation capabilities. Our comprehensive evaluation of various state-of-the-art commercial LLMs highlights the significant impact of feedback mechanisms and the utilization of in-context learning capabilities on model performance. The results underscore both the challenges and opportunities for LLMs in the PIC design domain, providing targeted insights for future research and development to advance automation and efficiency in this field.

Acknowledgment

This work is supported by the Nansha District Key Area S&T Scheme (No. 2024ZD007), Guangzhou Municipal Science and Technology Project (Guangzhou EDA Key Laboratory, No.2023A03J0013), and Natural Science Foundation of Guangdong Province (No.2024A1515012438).

References

[1] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avizienis, S. Lin *et al.*, "Single-chip microprocessor that communicates directly using light," *Nature*, 2015.

[2] K.-i. Kitayama, M. Notomi, M. Naruse, K. Inoue, S. Kawakami, and A. Uchida, "Novel frontier of photonics for data processing—photonic accelerator," *Apl Photonics*, 2019.

[3] K. Lu, Z. Chen, H. Chen, W. Zhou, Z. Zhang, H. K. Tsang, and Y. Tong, "Empowering high-dimensional optical fiber communications with integrated photonic processors," *Nature Communications*, 2024.

[4] Z. Xu, T. Zhou, M. Ma, C. Deng, Q. Dai, and L. Fang, "Large-scale photonic chiplet taichi empowers 160-tops/w artificial general intelligence," *Science*, 2024.

[5] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[6] Anthropic, "The claude 3 model family: Opus, sonnet, haiku."

[7] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[8] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024.

[9] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023.

[10] D. Vungarala, S. Alam, A. Ghosh, and S. Angizi, "Spicepilot: Navigating spice code generation and simulation with ai guidance," *arXiv preprint arXiv:2410.20553*, 2024.

[11] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "Chateda: A large language model powered autonomous agent for eda," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[12] R. Li, C. Zhang, S. Mao, H. Huang, M. Zhong, Y. Cui, X. Zhou, F. Yin, S. Theodoridis, and Z. Zhang, "From english to pcsel: Llm helps design and optimize photonic crystal surface emitting lasers," 2023.

[13] J. Liu, A. Sharma, C. Doumbia, and J. K. Poon, "Towards large-language model assisted layout of silicon photonic integrated circuits," in *European Conference on Integrated Optics*, 2024.

[14] F. Laporte, "Sax," 2023. [Online]. Available: https://github.com/flaport/sax

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.