

Large Language Models for EDA: Future or Mirage?

ZHUOLUN HE, The Chinese University of Hong Kong, Hong Kong, Hong Kong
YUAN PU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
HAOYUAN WU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
TAIRU QIU, ChatEDA Tech, Foshan, China
BEI YU, CSE, The Chinese University of Hong Kong, Hong Kong, Hong Kong

In this article, we explore the burgeoning intersection of large language models (LLMs) and electronic design automation (EDA). We critically assess whether LLMs represent a transformative future for EDA or merely a fleeting mirage. By organizing existing research into four critical domains of EDA—code generation, verification and debugging, knowledge representation and retrieval, and optimization/modeling—we provide a comprehensive overview of the current state-of-the-art. The survey concludes with a 5-level roadmap to guide the progressive integration and advancement of LLMs in EDA. Ultimately, this article aims to provide a comprehensive, evidence-based perspective on the role of LLMs in shaping the future of EDA.

CCS Concepts: • **Hardware → Electronic design automation**; • **Computing methodologies → Natural language processing;**

Additional Key Words and Phrases: Large Language Model

ACM Reference Format:
Zhuolun He, Yuan Pu, Haoyuan Wu, Tairu Qiu, and Bei Yu. 2025. Large Language Models for EDA: Future or Mirage?. *ACM Trans. Des. Autom. Electron. Syst.* 30, 6, Article 90 (October 2025), 53 pages. https://doi.org/10.1145/3736167

1 Introduction

The semiconductor industry is at the forefront of technological innovation, with modern **integrated circuits** (**ICs**) becoming increasingly complex in terms of both design and functionality. As the demand for higher performance, lower power consumption, and smaller form factors continues to rise, traditional **Electronic Design Automation** (**EDA**) tools are facing unprecedented challenges. These tools, which have been the backbone of chip design for decades, are now struggling to keep pace with the rapid advancements in technology and the growing intricacy of IC designs. The

This work is partially supported by The Research Grants Council of Hong Kong SAR (No. RFS2425-4S02 and No. CUHK14210723).
Authors' Contact Information: Zhuolun He, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: zleonhe@gmail.com; Yuan Pu, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: pyxch11081210@gmail.com; Haoyuan Wu, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: wuhyhowell@gmail.com; Tairu Qiu, ChatEDA Tech, Foshan, China; e-mail: m18826076901@gmail.com; Bei Yu (Corresponding author), CSE, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: byu@cse.cuhk.edu.hk.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).
ACM 1084-4309/2025/10-ART90
https://doi.org/10.1145/3736167

limitations of these conventional methods, such as long design cycles, high costs, and the need for extensive human expertise, underscore the necessity for new approaches.

Concurrently, the field of **artificial intelligence** (**AI**), particularly **machine learning** (**ML**), has witnessed remarkable progress. **Large Language Models** (**LLMs**), a subset of ML, have demonstrated exceptional capabilities in understanding and generating human-like text. This success has sparked interest in applying LLMs to a wide array of domains, including EDA, where they can potentially address some of the most pressing challenges faced by the industry.

LLMs [1–4] are sophisticated AI systems that are trained on vast datasets of text and code, enabling them to perform a wide range of natural language processing tasks [5–8]. In the context of EDA, LLM for EDA refers to the application of LLMs to various stages of the electronic design process, from initial architectural exploration to detailed physical design and post-silicon validation. The primary objective of LLM for EDA is to leverage the powerful language processing and pattern recognition capabilities of LLMs to enhance the efficiency, accuracy, and automation of EDA workflows. The integration of LLMs into EDA holds significant potential for transforming the way chips are designed. By automating repetitive and labor-intensive tasks, LLMs can accelerate the design process, reduce the likelihood of human error, and free up engineers to focus on more creative and strategic aspects of design. For example, LLMs can assist in generating and optimizing hardware descriptions, performing design space explorations, and even aiding in the debugging and verification of complex designs. Moreover, LLMs can provide a more intuitive and user-friendly interface, allowing designers to interact with EDA tools using natural language. This could democratize access to advanced EDA capabilities, making it easier for non-experts to engage in the design process. Additionally, the ability of LLMs to learn from large datasets and adapt to specific tasks through fine-tuning offers the potential for highly customized and optimized design solutions.

This survey provides a rigorous exploration of the integration of LLMs into four critical domains of EDA: code generation, verification and debugging, knowledge representation and retrieval, and optimization/modeling. These categories reflect the broad capabilities of LLMs in addressing critical challenges throughout the design and development pipeline. Code generation focuses on the automated synthesis of **hardware description languages (HDLs)**, system-level designs, analog circuits, **high-level synthesis** (**HLS**), and flow scripts, significantly accelerating prototyping and reducing manual effort. Verification and debugging encompass the use of LLMs to ensure design correctness, detect and resolve errors, and validate functionality and security through tasks like testbench generation, assertion synthesis, fault localization, and bug fixing. Knowledge representation and retrieval leverages LLMs to encode, retrieve, and reason over vast amounts of design-related data, enabling efficient specification analysis, bug classification, and **question answering** (**QA**) to support informed decision-making. Finally, optimization/modeling focuses on improving key design metrics such as **power, performance, and area** (**PPA**) by employing LLMs for tasks like circuit sizing, layout refinement, and workflow automation. This taxonomy not only underscores the versatility of LLMs in tackling various stages of the design flow but also elucidates the potential for improved designer productivity, enhanced design quality, and accelerated innovation.

Figure 1 illustrates some of the covered topics of this survey, centered around four principal axes of modality transformation: code-to-code, code-to-language, language-to-code, and language-to-language. The space is divided into four quadrants, respectively, named as the *compilation* quadrant (C2C), the *codegen* quadrant (NL2C), the *chatbot* quadrant (NL2NL), and the *modeling* quadrant (C2NL). Each axis represents a distinct approach to leveraging LLMs' prowess in interpreting and generating information, thereby bridging the gap between the abstract concepts of human intent and the concrete realities of hardware implementation. Through this lens, we will explore how

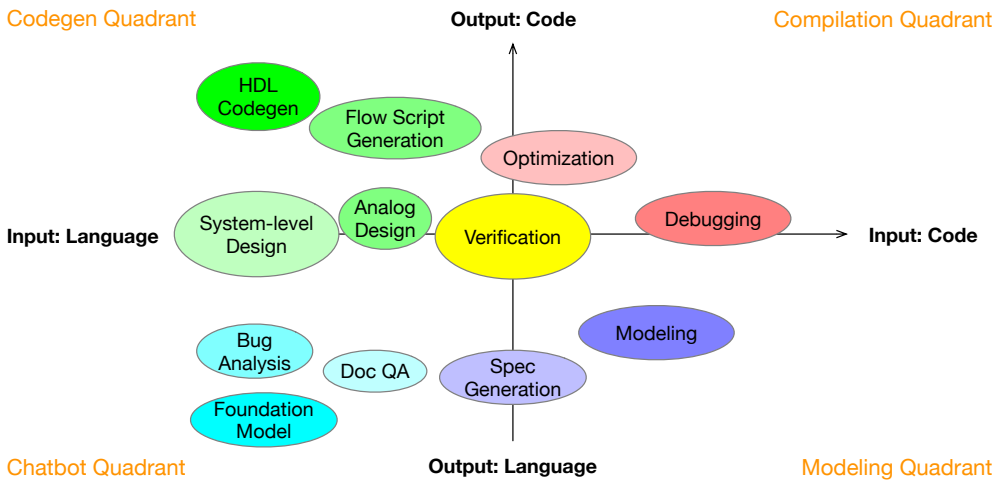


Fig. 1. Covered topics of this survey.

LLMs are reshaping EDA practices, from automating low-level coding tasks to enabling high-level design specification through natural language commands.

2 Preliminary

2.1 EDA

EDA is a category of software tools used for designing and verifying electronic systems, including ICs, **printed circuit boards** (**PCBs**), and other complex electronic components. EDA tools are essential for the semiconductor industry, enabling designers to create, simulate, and verify the functionality and performance of electronic designs before they are manufactured. The importance of EDA lies in its ability to manage the increasing complexity of modern electronic systems, reduce design time, and ensure that designs meet stringent PPA requirements.

EDA involves a comprehensive and iterative workflow that transforms high-level specifications into physical chip designs [9, 10]. The EDA process begins with the specification and architecture phase, where the system's functional requirements, performance targets, and constraints are defined. This is followed by the HLS or **register-transfer level** (**RTL**) design, in which the architectural specifications are translated into an HDL that represents the digital circuit at a higher abstraction level. Logic synthesis then converts the RTL code into a gate-level netlist, optimizing for area, power, and timing. The subsequent physical design stages, including placement and routing, focus on the spatial arrangement of the logic gates and the interconnections between them, respectively, to form the final layout. Throughout these stages, sophisticated algorithms and tools are employed to ensure that the design meets the specified performance criteria while adhering to manufacturing constraints. The integration of ML techniques into each of these EDA stages is increasingly providing new opportunities for automation, optimization, and enhanced design quality [11]. Figure 2 illustrates typical ML methodologies applicable to different EDA stages, which is extended from [12]. Great success has been achieved by applying appropriate ML methods to design stages and corresponding data, such as CNN for placement optimization [13], GNN for netlist representation learning [14], and CNN+GNN for timing prediction [15]. LLMs would be depicted as supporting tools that interact with the human designers and engineers, providing assistance in areas where natural language understanding and generation are beneficial. This integration aims to

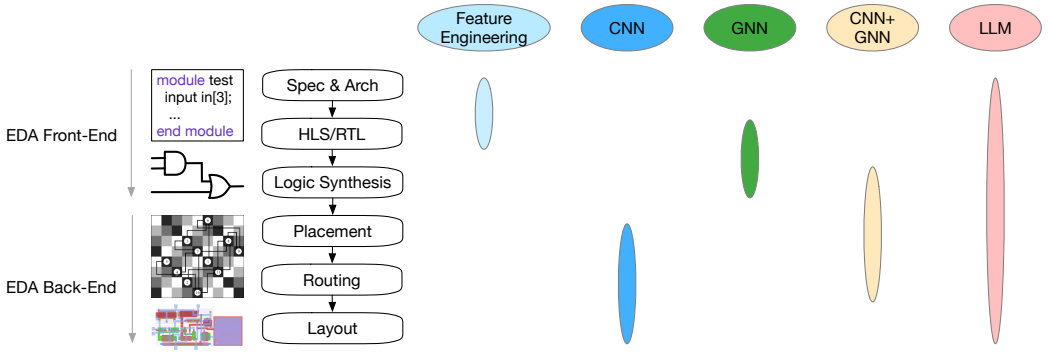


Fig. 2. Typical ML methodologies in EDA stages. Extended from [12].

improve the efficiency and effectiveness of the EDA process by automating routine tasks, enhancing decision-making, and facilitating better communication between the design teams and the EDA tools.

2.2 LLMs

LLMs represent a significant advancement in the field of **natural language processing** (**NLP**) and have become a cornerstone of modern artificial intelligence. These models are characterized by their massive scale, often containing billions or even trillions of parameters, and their ability to generate coherent, contextually relevant, and human-like text. The core innovation that has enabled this leap forward is the Transformer architecture [16]. Transformers rely on self-attention mechanisms, which allow the model to weigh the importance of different words in a sentence, thereby capturing long-range dependencies and complex contextual relationships more effectively than previous architectures. It is observed that language model log-likelihood loss scales consistently as a power-law with model size, dataset size, and the amount of compute used for training [17], which is known as the *scaling law* of LLMs. Scaling up language models has shown to unpredictably bring about their *emergent abilities* [18], which are not present in smaller models but is present in larger models.

Architectural Variants. Two primary architectural variants of transformers are commonly used in LLMs: *encoder-only* and *decoder-only*.

Encoder-only models, such as BERT (i.e., Bidirectional Encoder Representations from Transformers) [19], consist solely of the encoder part of the original transformer architecture. They are designed for tasks that require understanding the input sequence, such as text classification, named entity recognition, and QA. These models are trained bidirectionally, meaning they can consider context from both the left and right sides of a token. A common pre-training task for encoder-only models is **masked language modeling** (**MLM**), where some tokens in the input are masked, and the model is trained to predict the original values of those tokens based on the surrounding context.

On the other hand, decoder-only models, such as GPT-4 (i.e., Generative Pre-trained Transformer) [1], use only the decoder part of the transformer architecture. They are well-suited for generative tasks, such as text completion, story writing, and code synthesis. These models are trained unidirectionally, predicting each token in a sequence based on the preceding tokens. This *autoregressive* nature makes them powerful for generating coherent and contextually relevant text. A typical pre-training task for decoder-only models is **causal language modeling** (**CLM**), where the model is trained to predict the next token in a sequence given the previous tokens. Wang et al.

[20] have presented a large-scale evaluation of modeling choices and their impact on zero-shot generalization.

Training Methods. Training LLMs involves several sophisticated techniques to ensure that the models can effectively learn from large datasets and generalize to a wide range of tasks. The process typically begins with *pre-training*, which is the initial phase where the model learns generalizable representations of language from a large corpus of text data. For encoder-only models, this often involves MLM, while for decoder-only models, it includes CLM. Given a sequence of tokens $x = \{x_1, \ldots, x_n\}$, the CLM task aims to autoregressively predict the target tokens x_i based on the preceding tokens $x_{<i}$ in a sequence. A general training objective is to maximize the following likelihood:

$$\mathcal{L}_{CLM}(x) = \sum_{i=1}^{n} \log P(x_i \mid \mathbf{x}_{<i}). \tag{1}$$

The goal of pre-training is to equip the model with a broad understanding of language that can be fine-tuned for specific tasks.

After pre-training, the model undergoes ***supervised fine-tuning*** **(SFT)**, which adapts the pre-trained model to perform well on particular NLP tasks. During SFT, the model is trained on a smaller, task-specific dataset, allowing it to specialize in the target application. This phase is crucial for tailoring the model's capabilities to the specific requirements of the task, such as sentiment analysis, machine translation, or code generation.

Finally, alignment is a critical step in ensuring that the model's outputs are not only accurate but also aligned with human preferences and ethical standards. Alignment training aims to guide the model to produce responses that are safe, useful, and reflective of human values. Techniques such as **reinforcement learning from human feedback** **(RLHF)** [21] and supervised alignment, where the model is trained on a dataset of human-generated examples, are employed. Alternative optimization techniques like **direct preference optimization** **(DPO)** [22] are proposed. Alignment ensures that the model generates high-quality, contextually appropriate, and ethically sound content.

Parameter-efficient fine-tuning **(PEFT)** aims to reduce the number of trainable parameters while retaining a good performance as possible. LoRA [23] and its quantized counterpart QLoRA [24] are widely adopted Low-Rank Adaptation techniques, which freeze the pretrained model weights and inject trainable rank decomposition matrices into each layer of the Transformer architecture. The reparametrization can be described as

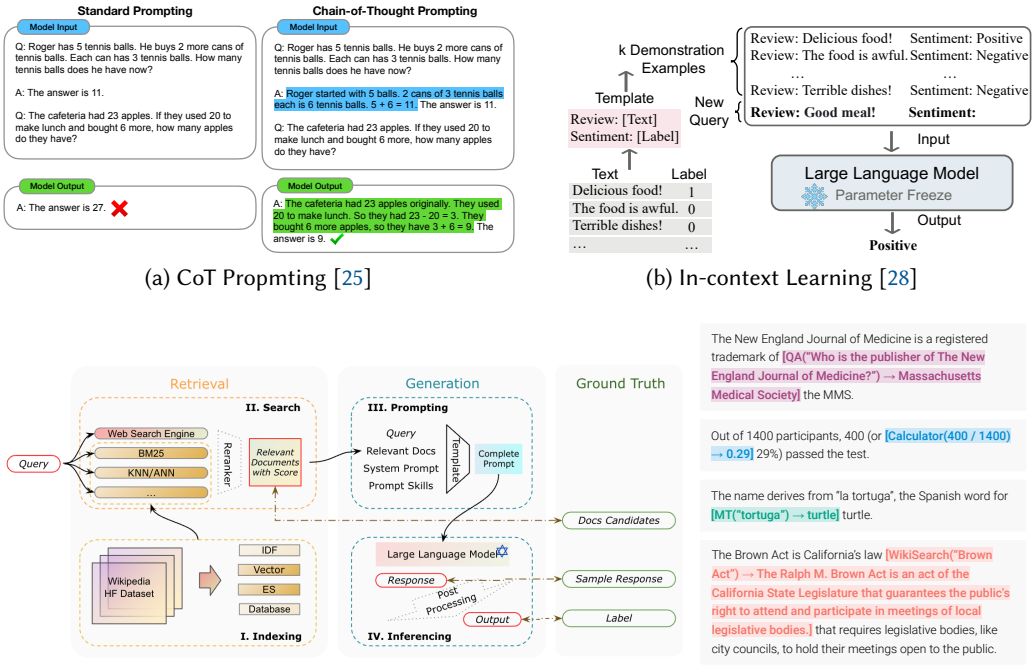
$$h = W_0 x + \Delta W x = W_0 x + BAx, \tag{2}$$

where W_0 is the pretrained weight matrix, $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and $r \ll \min(d, k)$.

2.3 LLM Augmentation

Augmenting LLMs involves enhancing their capabilities through various techniques so that the model's performance, adaptability, and ability to handle complex tasks are improved. Below, we describe some of these augmentation techniques, including **chain-of-thought** **(CoT)** prompting, **retrieval-augmented generation** **(RAG)**, and tooling agents.

CoT Prompting. CoT Prompting [25] has gained significant attention due to its effectiveness in improving the reasoning abilities of LLMs. CoT prompting involves providing the model with a series of thought processes or logical steps that lead to a solution, rather than just presenting it with a problem and expecting an immediate answer. This method helps the model to develop a more structured and coherent way of thinking, allowing it to tackle complex questions that require multi-step reasoning, such as complex arithmetic, logical puzzles, and commonsense reasoning. It not only enhances the quality of the model's output but also increases the transparency of its



(a) CoT Propmting [25] (b) In-context Learning [28]

(c) RAG Workflow [29] (d) Tooling Agents [30]

Fig. 3. Approaches to enhance LLM performance.

decision-making process, making it easier for users to understand and trust the generated solutions. Other prompting techniques like **in-context learning** (**ICL**) [26] are explored. Figure 3(a) and (b) illustrates CoT prompting and ICL, respectively.

RAG. RAG [27] combines the strengths of retrieval-based systems, which are adept at finding relevant information from a large corpus, with the generative capabilities of LLMs. In this framework, the model first retrieves relevant documents or pieces of information from a database and then uses this retrieved context to inform and improve the quality of the generated output. The integration of external knowledge not only enriches the content produced by the LLM but also ensures that the information is up-to-date and factually accurate, as the model can rely on a constantly updated source of information rather than being limited to its pre-training data. This dual mechanism of retrieval and generation makes RAG a powerful tool for tasks requiring both broad knowledge and precise detail. Figure 3(c) illustrates RAG workflow.

Tooling Agents. Tooling agents [30] represent a sophisticated approach to augmenting the capabilities of LLMs by equipping them with access to external tools and services, such as APIs, databases, and specialized software, to gather, process, and utilize information. This integration allows LLMs to perform tasks that go beyond their inherent knowledge and computational limitations, thereby expanding their functionality and utility in real-world applications. The architecture of a tooling agent typically includes the LLM core, which is the pre-trained language model responsible for generating text and understanding natural language, and a tool interface layer, which acts as a middleware to facilitate communication between the LLM and the external tools. This layer handles API calls, data retrieval, and response formatting. Additionally, there is an orchestration mechanism that coordinates the interaction between the LLM and the tools, ensuring that the right tools

are called at the appropriate times and that the results are integrated seamlessly into the LLM's responses. Figure 3(d) illustrates examples of tooling agents.

3 Code Generation

Code generation is a critical component of EDA, encompassing the automated creation of design artifacts such as HDL code, flow scripts, HLS outputs, and analog circuit descriptions. The application of LLMs to code generation offers significant opportunities to streamline these processes by automating repetitive tasks, generating complex code structures, and improving design efficiency. LLMs have the potential to translate high-level design specifications into HDL code, automate the creation of flow scripts for design and verification processes, generate HLS outputs from abstract functional descriptions, and even assist in analog circuit generation by producing parameterized templates. By leveraging their extensive training on diverse datasets, LLMs can reduce development time, improve accuracy, and enable the exploration of new design paradigms. This section provides a detailed analysis of the role of LLMs in code generation, their current capabilities, and the challenges and opportunities associated with their adoption in EDA workflows.

3.1 HDL Generation

LLM-driven HDL generation leverages the sophisticated understanding and generative capabilities of these models to translate high-level design specifications into detailed RTL code, the foundational layer in digital circuit design. This process begins with the input of functional requirements or behavioral descriptions, often articulated in natural language or pseudo-code, which the LLM deciphers with unparalleled precision. The model then synthesizes this information, applying its deep knowledge of hardware design principles and RTL syntax, to output syntactically correct and logically sound RTL code.

Recent studies measure code functional correctness through the pass@k metric [7], where a problem is considered solved if any of the k samples pass the unit tests. An unbiased estimation is given by

$$\text{pass@k} := \mathop{\mathbb{E}}_{\text{problem}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \tag{3}$$

where $n \geq k$ samples are generated per task, in which $c \leq n$ samples pass testing.

DAVE [31] investigates "English to Verilog translation" by a pretrained language model GPT-2. It targets a template-based, random auto-marking dataset, which are straightforward for translation. DAVE returned the correct answer in 94.8% of all validation tests, while bearing limitations like it cannot generate signal names that are not described.

Tharkur et al. [32] characterize the ability of LLMs to generate useful Verilog. They curate Verilog datasets comprising of open-source Verilog code in public GitHub repositories as well as Verilog textbooks, which are used to fine-tune five pre-trained LLMs for evaluation. They report that fine-tuning LLMs over a specific language clearly improves completion performance: pre-tuned LLMs produced completions that are functionally correct only 1.09% of the time, while this number increases to 27.0% after tuning. A more comprehensive analysis has been reported in Verigen [33].

Tomlinson et al. [34] explore designing a Spiking Neuron Array using LLM. Vitolo et al. [35] investigate designing a recurrent spiking neural network while also generating test benches. Both implementations are submitted for fabrication in the SkyWater 130 nm technology.

Sun et al. [36, 37] propose to first let LLMs classify the type of the circuit into combinational logic (*COMB*), sequential logic (*SEQU*), and a general type (*BEHAV*), based on the specifications. The original design specification is also transformed into a specific information list, such as state-transition table for sequential logic and truth table for combinational logic. LLMs then generate

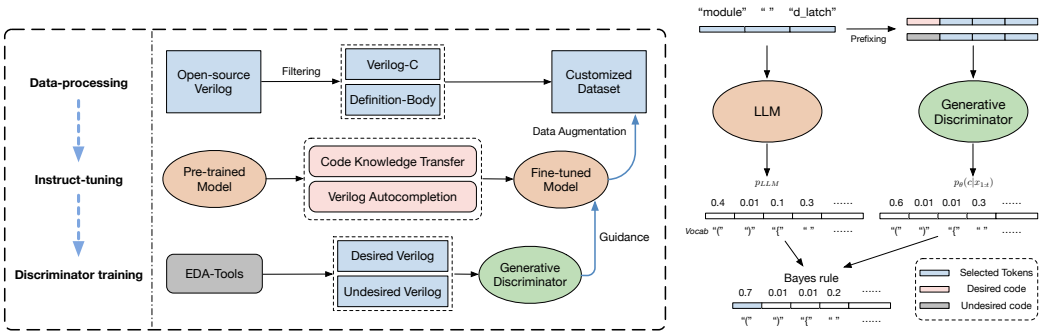


Fig. 4. Generative discriminator guidence in BetterV [40]. (a) Training overview. (b) An illustrative example.

the codes by following specific design paradigms depending on the circuit type. Experiments on VerilogEval show that the proposed method outperforms the GPT4 baseline in all Pass@k settings, with up to 14.7% in VerilogEval-human Pass@10.

MEV-LLM [38] integrates multiple LLMs, each specifically fine-tuned with dataset of a different complexity level. A complexity classifier LLM first assesses design complexity from the description prompt, and then forwards the task to the selected expert. Experiments show that the new multi-expert approach improves Verilog code generations by up to 23.9% using pass@k metrics compared with previous state-of-the-arts.

Authors of HDL-GPT [39] hypothesize that training data (HDL code) quality is the key to create models with exceptional performance. They propose a data augmentation approach using CoT prompting, pipelining task-specific steps, such as language identification, adding comments, and testbench generation. The final validation stage compiles the code samples through the full EDA flow to GDS generation. HDL-GPT2 (fine-tuned StarCoder2) delivered the highest performance on VerilogEval, with 60.6% at pass@1, 78.5% at pass@5, and 81.4% at pass@10 on human eval, and 67.0% for pass@1, 86.0% for pass@5, and 90.2% for pass@10 on machine eval.

BetterV [40] is a Verilog generation framework that performs design optimization through controlled code-to-code transformations. It integrates LLMs with discriminative guidance, enabling the system to generate Verilog code optimized for specific tasks. During the code generation process, the discriminator weighs the likelihood of the next token (code piece) being generated, based on both the language model's prediction and the conditional probability of the token conforming to the desired design characteristics. Figure 4 illustates the guidance from generative discriminator. The pass@1 on VerilogEval-machine and VerilogEval-human for BetterV outperforms GPT-4 by 8.1% and 2.6%, respectively. It also has 46.52% fewer nodes in a 'synthesis node reduction' task, and saves 22.45% in another 'verification runtime reduction' task.

In CodeV [41], authors argue that LLMs excel in summarizing Verilog code rather than generating them. Therefore, they propose to collect high-quality real-world Verilog data from Github, and then annotate them with LLMs. They introduce a multi-level summarization method, guiding LLMs (with few-shot examples) in first producing detailed descriptions and then abstracting high-level summaries. CodeV-CodeQwen achieves 77.6% pass@1 on the VerilogEval- machine benchmark and 53.2% on the VerilogEval-human benchmark, outperforming GPT-4 and previous SOTA model BetterV, and also achieves a 93.1% syntax pass rate and 55.2% function pass rate on the RTLLM benchmark, outperforming previous SOTA model RTLCoder.

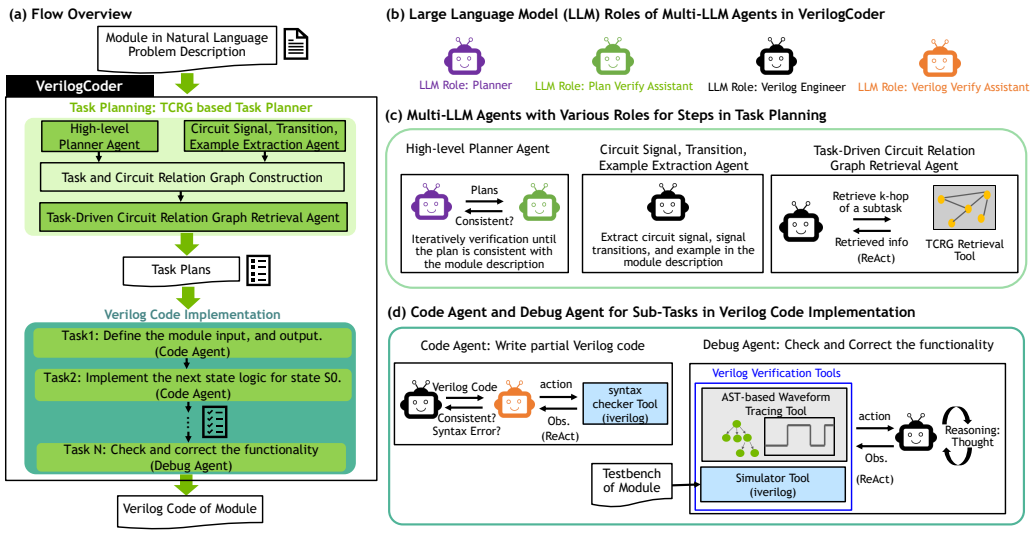


Fig. 5. Overall flow of multi-agent VerilogCoder [42].

VerilogCoder [42] explores the use of multiple AI agents for autonomous Verilog code completion. A customized task planner is introduced, which is based on *Task and Circuit Relation Graph* that captures step-by-step sub-tasks and related circuit information (signal transition, etc.). A set of Verilog tools, such as syntax checker and AST-based waveform tracing tool, are utilized to assist agents for code implementation. Figure 5 illustrates the overall flow of VerilogCoder. The proposed VerilogCoder achieve 94.2% pass rate on VerilogEval-Human v2, including syntax and functional correctness, and outperform one of the state-of-the-art methods by 33.9%, i.e., without an agent-based approach.

HaVen [43] presents a specialized framework for Verilog code generation that mitigates hallucinations and aligns LLM outputs with HDL engineering practices. HaVen addresses three categories of hallucinations—symbolic, knowledge, and logical—through a structured methodology. It incorporates **Symbolic Interpretation-based Chain-of-Thought (SI-CoT)** to accurately translate inputs such as truth tables and state diagrams into natural language instructions, a knowledge-enhanced dataset (K-dataset) to improve adherence to digital design conventions and Verilog-specific attributes, and a logical-enhanced dataset (L-dataset) to strengthen logical reasoning and handling of corner cases. These techniques are integrated into the framework to fine-tune LLMs, significantly improving Verilog code generation performance. Experimental results on benchmarks like VerilogEval and RTLLM demonstrate up to a 6.7% increase in pass@1 and a 4.7% increase in pass@5, outperforming state-of-the-art models.

Fang et al. [44] introduce the use of LLMs for data augmentation in MasterRTL, a presynthesis PPA estimation framework for RTL designs. In the LLM-based data augmentation strategy, they generate hierarchical RTL designs by decomposing high-level design requirements into submodules and creating scalable, synthesizable Verilog code. This iterative process involves automated error detection and correction using synthesis feedback, ensuring the generated designs meet functional and structural requirements. The LLM-based approach complements a graph-based data augmentation method, enhancing diversity and hierarchy in the synthesized RTL dataset. By leveraging LLMs, MasterRTL effectively addresses the scarcity of high-quality RTL designs, improving the training of ML models for accurate PPA predictions and enabling transferable modeling across various design stages and technology nodes.

Prompt Augmentation. Vijayaraghavan et al. [45] propose a **Chain-of-Descriptions (CoDes)** scheme to enhance HDL generation performance by outlining structured intermediate descriptive steps. The process first formulates a plan that breaks down the task into manageable steps. Next, it refines LLM responses by removing unnecessary boilerplate text and ensures style consistency. Finally, the refined plan is integrated with the input prompt to generate the LLM's response. The introduction of intermediate descriptive steps enhances Granite-based models for a 40% improvement in the Pass@1 metric on testbenches.

AutoVCoder [46] introduces an RAG mechanism that retrieves examples and knowledge from a domain-specific database. The retriever is trained on contrastive sample pairs, where a positive sample pair is formed when the addition of a document chunk enhances the output quality of an LLM for a given question. It shows a 0.5% and 2.2% improvement in functional correctness on the EvalMachine and EvalHuman benchmarks compared with BetterV, and achieves an increase in syntactic and functional correctness of up to 3.4% on the RTLLM benchmark compared with RTLCoder.

Nair et al. [47] discuss prompt design techniques for secure hardware generation in scenarios listed in **Common Weakness Enumeration** (**CWE**).

Feedback-in-the-loop. Autochip [48] aims for a fully automated approach that removes the designer from the loop. It enhances Verilog designs by automatically identifying and rectifying compilation errors and functional bugs in multiple rounds of interaction with a baseline LLM. In particular, outputs from compilation and debug tools with testbench simulations are collected to prompt the LLM to refine its implementation. Autochip improves the success rate against functional testbenches by on average 24.2% w.r.t. baseline generation.

ChipGPT [49] includes both machine and human feedback in the iterative chip design flow. A *correction manager* checks program correctness using compiler and simulator, followed by an auxiliary human feedback stage. Chang et al. also design a post LLM search approach [50] to accomodate a **design space exploration** (**DSE**) process. EDA tools are employed to evaluate the PPA of each HDL program enumerated in a defined search tree. It has an area reduction of 47% compared with the original ChatGPT solution.

Origen [51] introduces a self-reflection loop that comprises a base LLM with two trained LoRA models: *Gen LoRA* and *Fix LoRA*. *Gen LoRA* is responsible for interpreting natural language instructions and generating initial RTL code, which then undergoes compilation evaluation. If the code fails to pass evaluation, *Fix LoRA* analyzes the error messages and the erroneous code, proposes and implements fixes to address the identified issues, and the corrected code is re-evaluated. OriGen surpasses the previous best-performing LLM by 12.8% on the VerilogEval-Human benchmark and is comparable with GPT-4 Turbo, and surpasses GPT-4 by 19.9% in syntactic correctness on the VerilogFixEval benchmark.

AIvril [52] interweaves verification with the generative process, utilizing an *AutoReview* module and an *AutoDV* module. A review agent examines compilation logs for errors and generates structured review prompts to assist code refinement. Until syntax errors are resolved, the verification workflow begins and outputs coverage reports and failed assertions. The Review Agent similarly identifies discrepancies and formulates comprehensive review prompts. AIvril improves 1.32× and 2× in code quality on the VerilogEval-Human benchmark compared to CodeV and RTLFixer, respectively.

VeriAssist [53] generates a test bench with test cases alongside the RTL code, which enables self-verification by analyzing code behavior considering time constraints. It then performs self-correction by identifying and rectifying compilation errors and functional bugs given feedback from the simulator. VeriAssist improves 7.5% and 7.0% pass@1 scores on VerilogEval-Machine and VerilogEval-Human, and improves 10.4% on the functionality pass@5 score on RTLLM with a syntax pass rate of 100% using GPT-4.

VeriPPA [54] includes a dual-stage refinement protocol in the Verilog generation loop. The first stage leverages detailed error diagnostics from the simulator to rectify and evolve the Verilog code. The second stage further inspects PPA of the design by calling a commercial tool for logic synthesis. The design passes the PPA check if if is synthesizable and meets the PPA goal. Otherwise, both the design and its corresponding PPA report will be fed back to the LLM for refinement. VeriPPA achieves 62.0% (+16%) for functional accuracy and 81.37% (+8.3%) for syntactic correctness in Verilog code generation, compared to few-shot GPT4.

RTLSquad [55] is an LLM-based multi-agent system designed for interpretable RTL code generation and optimization. RTLSquad addresses the lack of decision interpretability in existing methods by dividing the design process into three stages: exploration, implementation, and verification and evaluation, each managed by specialized agent squads. Through collaborative communication and natural language exchanges, RTLSquad generates functionally correct RTL code, optimizes PPA (power, performance, and area) metrics, and provides detailed decision paths to justify design choices. Experiments conducted on the RTLLM dataset demonstrate that RTLSquad improves Pass@1 scores by up to 7.2% compared to baseline methods and achieves PPA metrics that match or exceed reference designs in 73.3% of cases.

Hierarchy. Nakkab et al. [56] introduce hierarchical prompting that facilitates step-wise design methodology. The source of architectural hierarchy may come from either human-driven planning (HDHP) or purely automated generation (PGHP). Either way requires a *hierarchy extraction* step at the beginning to list all submodules necessary for design implementation. Then, the list is iterated over and LLM is responsible for producing HDL for each submodule. Finally, LLM is requested to integrate all generated submodules into a top-level module.

ChatCPU [57] is an agile CPU design and verification platform powered by fine-tuned large language models LLMs to automate the end-to-end hardware design process. ChatCPU introduces a **processor description language** (**PDL**) for standardized CPU design specifications by encapsulating timing, functionality, pipeline architecture, and pin-to-pin interfaces, enabling automated HDL generation and the creation of a cycle-accurate reference model. Additionally, it incorporates a **module-interchangeable verification paradigm** (**MVP**), which uses co-simulation between modularized RTL designs and golden reference models to verify functionality at both module and system levels. Leveraging fine-tuned LLMs, ChatCPU was used to design Saber6, a 6-stage in-order RISC-V CPU with 80,000 gates, successfully taped out using the SkyWater 130 nm MPW project. The platform achieved significant efficiency gains, including up to 12× acceleration in RTL coding, 9.33× in verification, and an average 3.81× reduction in design iteration time.

HiVeGen [58] introduces a hierarchical LLM-based Verilog generation framework designed to address challenges in scalable and complex chip design, particularly for **Domain-Specific Accelerators** (**DSAs**). HiVeGen overcomes the limitations of traditional LLM-generated monolithic HDL code—such as token limits, code redundancy, and high error-correction costs—by decomposing designs into manageable hierarchical submodules. The framework integrates a Hierarchy-Aware Prompt Generation Engine for DSE and hierarchical decomposition, a Weight-Based Retrieving Engine to enhance module reuse and reduce redundant code generation, and an On-the-Fly Parsing Engine for real-time code feedback and error correction. HiVeGen also supports PPA-aware (Power, Performance, Area) optimization through ICL templates, enabling the generation of high-quality, reusable, and efficient designs. Experimental results demonstrate up to 30.97% token savings, 45.24% runtime reduction, and improved generation accuracy compared to existing methods, while successfully generating PPA-optimized DSA designs such as Systolic Arrays, CGRAs, and ShiDianNao accelerators.

Reinforcement Learning (RL). Wang et al. [59] formulate Verilog program generation as an RL problem, where the rewards are defined by calculating the similarity between the AST of the response and the AST of the ground truth code. The authors argue that auto-regressive training is not inherently aligned with the parallel semantics of Verilog code, and that a well-defined global supervisory signal is critical. It achieves syntax pass@5 of 93.1% and functional pass@5 of 55.2% on RTLLM.

Search-based. Delorenzo et al. [60] present an automated transformer decoding algorithm that integrates **Monte-Carlo Tree Search** (**MCTS**). Specifically, in the MDP, the *action* is the vocabulary set of tokens of LLM, and the *reward* is determined based on the syntactical correctness, functional correctness, and PPA metric (the inverse area-delay product, larger the better) of the design. Comment line tokens are pruned to reduce the search space. Module reuse is also enabled so that LLMs can reference the generated sub-modules in their large-module generation, providing a more feasible MCTS iteration time. The MCTS technique can achieve a 31.8% improvement in the area-delay product for a 16-bit adder design generated by VeriGen.

Multi-modality. Chang et al. [61] present a multi-modal generative model tailored for Verilog synthesis from visual-linguistic inputs. They argue that natural language is insufficient for accurately conveying the nested spatial structures of hardware as well as multi-module hardware design. Therefore, a multi-modal (figure+language) solution is proposed, which reduces token cost while enhancing the quality of the generated code. A hierarchical benchmark and related infrastructure captured in a query language, named VLMQL, are also proposed. With the integration of visual inputs, both syntax correctness and functionality correctness are greatly improved.

Benchmark and Dataset. VerilogEval [62] is a benchmarking framework designed to evaluate the performance of Verilog code generation for hardware design and verification. The framework comprises a comprehensive evaluation dataset made up of 156 problems sourced from the Verilog instructional website HDLBits, covering a range from simple combinational circuits to complex finite state machines. Pinckey et al. [66] extend VerilogEval to support specification-to-RTL tasks in addition to the original code completion task. They also incorporate ICL prompts and a failure classification mechanism into the dataset, which allows for prompt engineering and failure analysis.

VHDL-Eval [67] translates the programs in VerilogEval into VHDL, and enriches the dataset with publicly available VHDL problem sets from tutorials. Self-verifying testbenches are handcrafted for all the problems, adhering to the template of the VUnit Testing framework.

RTLLM [63] proposes a comprehensive open-source benchmark dedicated to the automatic design RTL generation with natural languages. They summarize three progressive goals of the task, namely *syntax goal* (correctly synthesizable), *functionality goal* (passing test cases), and *quality goal* (good PPA). RTLLM has collected 30 common designs, including 11 arithmetic designs and 19 logic designs. Each design in the benchmark consists of a natural language description of design functionality, a testbench, and a hand-crafted Verilog implementation. OpenLLM-RTL [68] introduces the updated RTLLM 2.0, which is augmented to 50 hand-crafted designs.

RTL-Repo [74] is a benchmark for Verilog code completion. Verilog repositories are collected via Github API, and random, nonempty, non-comment lines of code are randomly selected to be the target line for model prediction. RTL-Repo contains 4,098 samples, ranging from 2K to 128K tokens, and each sample contains the context of the entire repository, the code of the current file being edited, excluding the target line and all lines after it, and the target line itself. *Exact Match* and *Edit Similarity* are used as evaluation metrics for the benchmark.

PyHDL-Eval [73] is a framework for evaluating LLMs on specification-to-RTL tasks in the context of Python-embedded **domain specific languages** (**DSLs**). It includes 168 problems, each containing Verilog reference solutions, Verilog test benches, Python test scripts, and workflow orchestration

scripts. The RTL design problems belong to 19 different categories, covering both combinational logic (e.g., mux and adder) and sequential logic (e.g., counter and register file).

Authors of RTLCoder [64] have introduced a training dataset generation flow composed of three stages: RTL domain keywords preparation, instruction generation, and reference code generation. In the first stage, commonly used logic components (adder, FSM, etc.) are generated by traversing a category tree, forming a keyword pool. Then, keywords are sampled and fed to GPT for instruction generation, namely to generate a specific design description for the target component. Finally, implementation code is automatically generated by prompting GPT. By going through all three stages and extra validation procedures, a total of 27, 000 data samples are generated in their work for model training. OpenLLM-RTL [68] introduces the extended RTLCoder-data dataset of 80K instruction-code data samples.

MG-Verilog [72] is a *multi-grained* Verilog dataset that encompasses descriptions at various levels of detail and corresponding code samples. The dataset consists of over 11, 000 Verilog code samples, each with natural language descriptions ranging from high-level summaries to detailed, line-by-line comments.

Goh et al. [71] have published a labelled Verilog dataset of 68, 122 data entries, where each entry contains a single module, followed by an LLM-generated description of the module.

Authors of AI-HDLCoder [70] release an AI-HDLCoder-Dataset with over 100,000 rows of VHDL code, which are mainly collected from online sources and deduplicated using Jaccard similarity and Min-Hash techniques.

SA-DS [75] is a systolic array-based accelerator dataset comprising spatial array designs following the standardized Berkeley's Gemmini accelerator generator template. Each data point in SA-DS includes a natural language description of the accelerator's micro-architecture and a Chisel description, facilitating LLM-based hardware accelerator design by few-shot learning.

Fu et al. [69] propose to utilize version control data from hardware projects to construct semi-synthetic dataset. The reason is that version control serves as a database that captures snapshots of a project at any given moment and offers a precise explanation of the project's evolution. The 5W1H (Who, What, When, Where, Why, How) journalistic principles serve as a framework to describe the modification of code.

In CraftRTL [65], Liu et al. present an analysis of fine-tuned LLMs on synthetic data. They report that models underperform on problems involving non-textual input formats, such as K-map and waveforms. Therefore, a correct-by-construction solution is proposed to deal with non-textual representations. To mitigate "minor" errors produced by LLMs, they also construct a debug dataset by injecting identified errors into correct code snippets.

MetRex [76] introduces a dataset for evaluating the ability of LLMs to reason about post-synthesis metrics of Verilog HDL designs, such as area, delay, and static power. The MetRex dataset comprises 25, 868 Verilog designs annotated with their corresponding post-synthesis metrics, alongside a CoT prompting template to improve LLM reasoning by breaking down metric estimation into logical steps. The authors demonstrate that **supervised fine-tuning** (**SFT**) on MetRex significantly improves the accuracy of LLM-based metric estimation, with gains of 37.0%, 25.3%, and 25.7% for area, delay, and static power, respectively, compared to few-shot prompting techniques. Compared to traditional regression-based methods, the fine-tuned LLM achieves 17.4% higher accuracy within a 5% error margin and 1.7× faster runtime by eliminating preprocessing overheads. While LLMs show promise for metric reasoning in smaller designs, challenges remain for complex designs and advanced metrics like dynamic power.

Table 1 shows the summary of LLM-driven RTL generation methods on VerilogEval and/or RTLLM. This table highlights the diversity of approaches and the growing success of fine-tuned and specialized LLMs in RTL generation tasks.

Table 1. Summary of LLM-driven RTL Generation Methods on VerilogEval and/or RTLLM

Work	Model			Method			VerilogEval		RTLLM	
	Backbone	Size	FT?	FB	RAG	MA	H	M	Syn	Func
VerilogEval [62]	GPT-3.5						45.8	69.1		
RTLLM [63]	GPT-4								81	50.0
RTLCoder [64]	DeepSeek-coder	6.7B	✓				50.1	76.5	93.1	48.3
Tharkur et al. [32]	CodeGen	16B	✓				43.9	52.6	86.2	24.1
Sun et al. [36]	GPT-4						69.6	81.4		
MEV-LLM [38]	CodeGen	16B	✓			✓	55.7	63.6		
HDL-GPT [39]	StarCoder2	15B	✓				78.5	86.0		
CodeV [41]	CodeQwen1.5	7B	✓				65.1	88.2	93.1	55.2
Wang et al. [59]	DeepSeek-coder	6.7B	✓						95.2	48.5
AutoVCoder [46]	CodeQwen1.5	7B	✓		✓		55.9	79.9	100	51.7
Origen [51]	DeepSeek-Coder	6.7B	✓				60.1	82.4		65.5
AIvril [52]	Claude 3.5 Sonnet			✓		✓	70.5@1			
VeriAssist [53]	GPT-4			✓			62.8	78.3	100	75.9
BetterV [40]	CodeQwen	7B	✓				53.7	79.4		
CraftRTL [65]	StarCoder2	15B	✓				72.4	86.9	93.9	65.8
HaVen [43]	DeepSeek-Coder	6.7B	✓				64.2	84.5	92.8	66.0

Table 2. Summary of Benchmarks and Datasets for HDL Generation

Name	#Samples	Source			Components		
		Coll.	Trans.	Syn.	Spec	RTL	TB
VerilogEval [62, 66]	156	✓	-	-	✓	✓	✓
VHDL-Eval [67]	202	✓	✓	-	✓	✓	✓
OpenLLM-RTL (RTLLM 2.0) [68]	50	✓	-	-	✓	✓	✓
OpenLLM-RTL (AssertEval) [68]	18	✓	-	-	✓	✓	-
OpenLLM-RTL (RTLCoder-Data) [68]	80K	✓	-	-	✓	✓	-
Semi-Syntectic Dataset [69]	6,546	✓	-	-	✓	-	-
CraftRTL [65]	110k	-	-	✓	✓	✓	-
AI-HDLCoder [70]	100,000 rows	✓	-	-	-	✓	-
Goh et al. [71]	68,122	-	✓	-	✓	✓	-
MG-Verilog [72]	11,000+	✓	-	-	✓	✓	-
RTLCoder [64]	27,000+		-	✓	✓	✓	-
PyHDL-Eval [73]	168	-	-	✓	✓	✓	✓
RTLLM [63]	30	✓	-	-	✓	✓	✓
RTL-Repo [74]	4,098	✓	-	-	-	✓	-
SA-DS [75]	9,216	-	-	✓	✓	✓	-
MetRex [76]	25,868	✓	-	-	-	✓	-

Table 2 summarizes the benchmarks and datasets for HDL generation.

Evaluation. CreativEval [77] presents a framework to evaluate the creativity of LLMs for hardware design generation. Four cognitive subcategories of creativity, namely, *fluency*, *flexibility*, *originality*, and *elaboration*, are quantified through various prompts and analytical metrics. According to the evaluation of their creativity metric and its components, GPT-3.5 achieves the highest result (surpassing GPT-4).

FormalEval [78] introduces a formal evaluation method designed to automatically assess the functional correctness of code generated by LLMs without the need for manual test case creation. It utilizes a **Sequential Equivalence Checking** (**SEC**) engine, which is capable of automatically and comprehensively checking whether the implemented code behaves identically to the specified one across all possible input scenarios. If the two programs are functionally equivalent, FormalEval returns a "Proven" statement; if not, it provides a "Falsified" statement along with a counterexample, which is an input that causes a behavioral discrepancy. Additionally, the authors propose two approaches to integrate FormalEval with LLM training: an explicit neural network-based functional checking proxy and an implicit functionality loss signal during the learning process. They reported a 23% evaluation error in existing RTL benchmarking dataset by using their method.

SimEval [79] introduces a comprehensive multifaceted metric designed to assess the similarity of HDL codes generated by LLMs. The metric evaluates similarity at three levels: syntactic (using abstract syntax trees), semantic (based on control flow graphs), and functional/structural (via synthesized gate-level netlists). SimEval leverages techniques such as Jaccard similarity for AST analysis, graph kernels for CFG comparison, and shortest path kernels for netlist evaluation. The authors applied SimEval to assess dataset diversity in RTL codes generated by state-of-the-art LLM-based frameworks like RTLCoder, revealing significant issues with diversity across clusters of generated designs.

Watermarking. RTLMarker [80] discusses hardware watermarking that protects RTL code generated by LLMs. It employs a rule-based Verilog code transformation set that ensures the functional correctness of watermarked code while embedding robust and transparent watermarks into both the RTL code and the synthesized gate-level netlist. The framework includes three core modules: a watermark embedding module, a feature representation module, and a watermark detection module, with joint optimization to balance transparency and effectiveness. Evaluations on benchmarks like RTLLM and VerilogEval show that RTLMarker achieves over 95% accuracy at the RTL level and demonstrates resilience against variable name replacement attacks.

3.2 System-level Design

System-level design represents a high-level abstraction in the hardware development process, focusing on functional modeling, architectural exploration, and design space tradeoffs rather than direct RTL implementation. Unlike traditional HDL code generation, which emphasizes the precise, low-level implementation of hardware modules, system-level design deals with algorithm-to-hardware translation, hardware-software partitioning, and macro-level decisions, such as selecting appropriate architectures or optimizing communication protocols. The key challenge in LLM-driven system-level design lies in bridging the gap between functional specifications and synthesizable hardware descriptions while ensuring PPA tradeoffs are considered.

Yan et al. [81] propose a novel framework, **LLM-based Co-Design Accelerator** (**LCDA**), that LLMs to address the "cold start" problem in **software-hardware** (**SW-HW**) co-design for **deep neural networks** (**DNNs**). LCDA uses pre-trained LLMs, specifically GPT-4, as design optimizers, allowing the framework to utilize heuristic knowledge from prior research and bypass the random initialization typically required in RL or genetic algorithm methods. The framework co-designs DNN topologies and **Compute-in-Memory** (**CiM**) DNN accelerators, achieving a 25× speedup compared to the state-of-the-art NACIM method while maintaining comparable performance.

In GPT4AIGChip [82], authors investigate the limitations and capabilities of leveraging existing LLMs to generate AI accelerator designs. After analyzing their common failures, they propose an LLM-friendly hardware design template, emphasizing high modularity, decoupled module design, and deep design hierarchies to facilitate step-by-step design generation. A carefully enhanced

Table 3. The Conversation Flow Map in Chip-Chat [83]

Cont. T. ID	T. ID	Topic	# User Msgs	# Restart	# User Lines	# User Chars	# LLM Lines	# LLM Chars
-	00	Specification	22	10	45	5,025	498	44,818
-	01	Register specification	6	2	59	4,927	91	9,961
-	02	Shift registers and memory	5	5	65	5,444	269	9,468
-	03	Multi-cycle planning and ALU	7	2	103	7,284	243	10,148
-	04	Control signal planning	13	21	216	9,205	414	20,364
-	05	Control Unit state logic	12	11	216	9,898	742	21,663
-	06	ISA to ALU opcode	4	0	72	4,576	149	5,624
-	07	Control unit output logic	11	6	266	8,632	518	19,180
-	08	Datapath components	12	0	144	5,385	516	15,646
-	09	Python assembler	3	4	127	4,231	218	6,270
00	10	Spec. branch update	1	1	14	1,275	15	1,635
07	11	Control Unit branch update	2	2	98	3,743	101	3,969
08	12	Datapath branch update	2	0	25	888	20	726
11	13	Control Unit bug fixing	6	1	190	5,413	241	8,001
-	14	Memory mapped components	7	0	79	3,079	516	16,237
-	15	Shift Register bug fix	2	0	38	985	85	2,593
12	16	Datapath bug fixing and updates	6	0	116	2,979	128	4,613
14	17	Memory mapped constants	4	0	21	849	101	4,655
03	18	ALU optimization	1	0	2	98	32	1,368
TOTALS			125	65	1,896	83,916	4,897	206,939

prompt, reinforced with similar demonstrations selected from a demo library, facilitates ICL for LLMs.

Chip-Chat [83] considers tool and human feedback in the conversation flow. If errors occurred multiple times, different levels of human feedback would be provided, and LLMs are responsible for regenerating the solution. In a later version, they detail the use of conversational LLMs for an *8-bit accumulator-based microprocessor*. Given the fixed-size context window of LLMs, the conversation is broken into subtask *threads*, and threads are organized in a linear flow to design, implement, and optimize required circuit components. A *restart* mechanism is reported for suboptimal responses from models. According to their record, "the processor was built through a linear flow of 125 user messages across 18 topics in 11 conversation threads". Table 3 shows the conversation flow map in Chip-Chat [83]. Blocklove et al. [84] investigate how LLMs may be used for both design and test together.

OPL4GPT [85] explores the use of LLMs to generate hardware descriptions using both Verilog and C++ for various **System-on-Chip** (**SoC**) functional blocks. The study evaluates the suitability of each language for different applications, comparing RTL synthesis (Verilog) and HLS (C++). Through a modular bottom-up design methodology, the framework decomposes hardware design tasks into sub-modules, generates their code and testbenches via LLMs, and integrates them into top-level modules. A closed-loop feedback mechanism is used to address syntax and functionality errors iteratively. Case studies, such as RSA encryption and CNN accelerators, demonstrate that C++ excels in algorithm-intensive and data-parallel tasks, while Verilog performs better in fine-grained control, peripherals, and processors.

AiEDA [86] introduces an agentic AI design framework for automating the end-to-end digital ASIC design process, from architectural specifications to GDSII layout, leveraging Generative AI and LLMs. AiEDA integrates agentic workflows with open-source EDA tools and techniques like

prompt engineering, few-shot learning, RAG, and self-reflection to manage complex design tasks iteratively. The framework comprises four key stages: architecture design, RTL generation, netlist synthesis, and physical design, each driven by AI and feedback loops with human intervention options. The article demonstrates AiEDA's capabilities through the design of a low-power digital ASIC for **Keyword Spotting** (**KWS**), showcasing hardware optimization strategies for power, area, and accuracy tradeoffs.

Salcedo et al. [87] introduce leveraging Generative AI, specifically GPT-4, to design and verify a vector processor SoC using open-source tools. The authors developed the SoC, including all SystemVerilog hardware blocks, C firmware, and test cases, through GPT-4, employing iterative prompt engineering and refinement techniques. The design incorporates a vector coprocessor integrated with an RISC-V CPU via the Wishbone bus and supports vector instructions for integer, bitwise, scalar, and floating-point operations. A hierarchical, iterative design process was used, combining GPT-4-generated code with human validation and functional testing via PyMTL3. The SoC was fabricated using the OpenLane flow and Sky130 PDK, achieving a $1.32mm^2$ area and 231 mW power consumption. Key contributions include insights into effective GPT-4 prompting strategies, such as generate-and-merge and example-based prompting, to address challenges like context window limits and complex module generation.

3.3 Flow Script Generation

Flow script generation represents a critical aspect that significantly impacts the efficiency and automation of the entire design process. Flow scripts, which are essential for orchestrating the sequence of operations within EDA tools, dictate the steps from synthesis to place-and-route, simulation, and verification. Traditionally, these scripts have been crafted manually, requiring deep expertise and a meticulous understanding of the toolchain. However, with the advent of LLMs, there is now an opportunity to automate the creation of these complex and often repetitive scripts. LLMs can interpret high-level design requirements and generate corresponding flow scripts that are not only syntactically correct but also optimized for specific design goals. This capability not only reduces the time and effort required for script writing but also minimizes the potential for human error, leading to more reliable and efficient design flows. By integrating LLM-driven flow script generation into the EDA workflow, designers can focus on higher-level tasks, such as architecture exploration and performance optimization, thereby accelerating the development cycle and improving overall productivity.

ChatEDA [88] is an innovative framework that leverages a LLM to automate the EDA process. The primary objective of ChatEDA is to interpret natural language commands and generate executable scripts that can be used to drive EDA tools, thereby streamlining the design flow from RTL to **Graphic Data System Version II** (**GDSII**). ChatEDA is designed as an autonomous agent that can understand and respond to user requirements in natural language. It breaks down complex tasks into smaller, manageable sub-tasks and generates Python scripts to execute these tasks using EDA tools. At the core of ChatEDA is AutoMage, a fine-tuned LLM specifically adapted for EDA tasks. AutoMage is trained to understand and generate code that interacts with EDA tools, such as OpenROAD, through their APIs. The workflow consists of three main stages: task planning, script generation, and task execution. AutoMage interprets the user's natural language requirements, plans the necessary sub-tasks, and then generates the corresponding Python scripts. Figure 6 illustrates the overview of AutoMage powered ChatEDA. A later version [89] improves AutoMage through enriched training corpus, instruction tuning with explanation, and CoTs prompting. A comprehensive evaluation benchmark, ChatEDA-bench, is also developed, which comprises 50 distinct tasks spanning three distinct categories: simple flow calls, complex flow calls, and parameter flow calls. A multi-agent collaboration system [90] is further developed to ensure reliable and

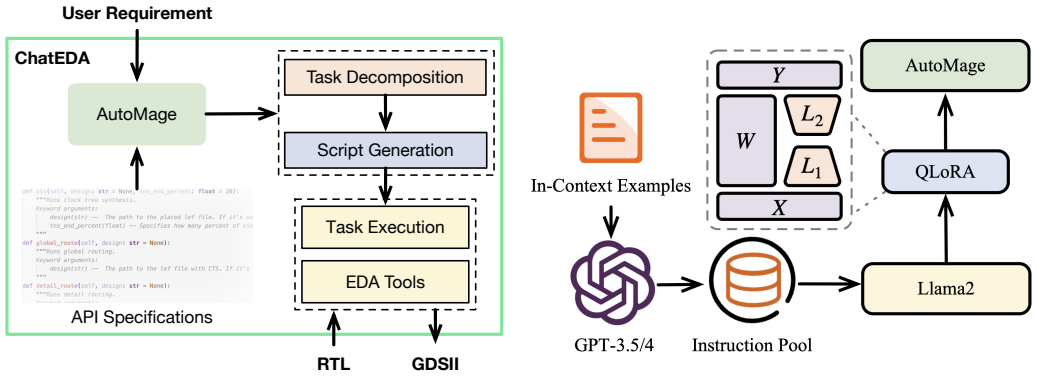


Fig. 6. AutoMage powered ChatEDA [88, 89]. (a) Overall workflow; (b) Instruction tuning.

successful EDA flow automation. Multiple 'divergent-thoughts agents' are responsible for diverse script generation, which are prompted by sampling few-shot examples from a demo pool. Then, a 'decision agent' determines the correct response.

ChipNeMo [91] presents a domain-adapted LLM for chip design that employs several domain adaptation techniques, including domain-adaptive tokenization, continued pretraining, and model alignment with domain-specific instructions. For EDA script generation, the methodology involves collecting production scripts from design experts and fine-tuning the LLMs on this dataset to generate both Python and Tcl scripts for VLSI timing analysis tasks. The process leverages ICL to improve the quality of the generated scripts, such as by adding inline comments, and integrates RAG to provide contextually relevant information during script generation, enhancing the model's ability to produce accurate and useful EDA tool scripts.

OpenROAD-Assistant [92] is an open-source LLM chatbot trained for physical design script generation and QA using OpenROAD (with separate QLoRA adapters). A retriever model is utilized to search for OpenROAD API list plus code templates for the prompt-script dataset, or OpenROAD documentation for the question-answer dataset. **Retrieval Augmented Fine-Tuning (RAFT)** is performed to reduce hallucination when the retriever model provides out-of-context data. OpenROAD-Assistant achieves a 77% pass@1 score, 80% pass@3 score for scripting, and it achieves a 98% BERTScore and 96% BARTScore on QA.

3.4 HLS

HLS code generation focuses on generating synthesizable HLS code, typically in C, C++, or SystemC, which can be directly processed by HLS tools to produce optimized hardware implementations. This step involves crafting high-level code that adheres to specific design constraints such as latency, throughput, power, and area (PPA), while ensuring compatibility with the synthesis process. Unlike traditional methods that involve manually writing HLS code, the integration of LLMs enables automated generation of synthesizable HLS code from functional descriptions, design requirements, or even natural language inputs. LLMs can assist in creating HLS code that incorporates proper hardware constructs, such as pipelining, loop unrolling, and parallelism, while ensuring the code is compliant with synthesis guidelines. By leveraging their ability to analyze and learn from large datasets of HLS examples, LLMs can not only generate code but also provide suggestions for optimization and best practices, drastically reducing development time and improving design quality.

C2HLSC [93] investigates LLMs refactoring C code into HLS-compatible formats. The process is broken down into small steps to allow the LLM to transform the original C into synthesizable C, such as removing print statements, optimizing data types using arbitrary width integers, and so on.

Xu et al. [94] similarly target an LLM-driven program repair framework that takes regular C/C++ code as input and automatically generates its corresponding HLS-C code for synthesis. A repair library containing correction templates for HLS is manually established. HLS-incompatible errors, such as the unsupported "pointer-to-pointer" feature, are identified and fixed with RAG-guided templates. The proposed LLM-driven repair framework can achieve much higher repair pass rates on 24 real-world applications compared with the traditional scripts and the direct applications of the LLM.

HLSPilot [95] is a framework that leverages LLMs to automate the generation and optimization of HLS code for hybrid CPU-FPGA architectures. The methodology begins with runtime profiling to identify performance bottlenecks in sequential C/C++ code. HLSPilot then extracts and optimizes these critical code segments using LLMs, which are guided by ICL and a set of predefined C-to-HLS optimization strategies. The optimized HLS code is further refined through DSE for pragma parameter tuning, and the final step involves offloading the compute kernels to the FPGA using Xilinx OpenCL APIs. This approach aims to bridge the semantic gap between high-level software and hardware design, enabling more efficient and automated hardware acceleration. In a case study involving the L-BFGS algorithm, HLSPilot achieved over 11 times speedup in runtime on a hybrid CPU-FPGA platform.

Swaroopa et al. [96] discuss LLM-based automatic generation of annotated C++ code suitable for HLS. They find that LLMs could not correctly deal with the K-maps problems.

Meech [97] case studies designing a Random Number Generator using Amaranth Python scripts.

Liao et al. [98] survey recent work in LLM for HLS, and they call for attention to power/energy implications in current research. The LLM solution reduces resource utilization for 38.67%, improves execution cycles for 64%, and reduces power consumption for 38.67%. For the critical path, the HLS-based approach outperformed LLM-based approaches by 28.82%.

SynthAI [99] is a multi-agent generative AI framework designed for the automated generation of modular HLS designs. SynthAI leverages advanced LLM technologies, including ReAct agents, CoT prompting, and RAG, integrated within a structured decision graph to decompose complex hardware design tasks into smaller, manageable modules. The framework includes agents for system design, module design, evaluation, and integration, which iteratively generate synthesizable HLS code, test benches, and header files while adhering to user-defined goals and requirements. SynthAI utilizes real-time web searches, database queries, and Python execution to enhance its design capabilities. Case studies, including the design of a 128-point FFT circuit, a quadruple-precision floating-point exponentiation module, and a simple RISC-V CPU, demonstrate SynthAI's proficiency in generating modular, synthesizable, and optimized HLS designs.

ChatChisel [100] explores generating Chisel code for RTL-level circuit design. Four LLM modules are utilized for design decomposition, Chisel code generation, tool-enhanced correction (i.e., debug), and top-level module composition, respectively. A five-stage RISC-V RV32I CPU has been designed using the ChatChisel workflow.

Gai et al. [101] examine the use of LLMs to automate hardware design generation through HLS. The authors address challenges such as limited HDL training data, semantic misalignment between pre-trained LLMs and HDLs, and the high token cost of HDL generation by leveraging HLS, which shares syntax and semantics with programming languages like C/C++. They introduce a dataset of over 42,000 HLS programs and fine-tune pre-trained models (e.g., Code-Llama) for HLS design generation, utilizing techniques like CoT prompting and feedback loops for iterative improvements. Results show gains in syntax correctness (from 54.85% to 94.33%) and functional correctness (from 0% to 61.45%), with generated HLS designs achieving reasonable performance on FPGA hardware.

3.5 Analog Design Generation

The integration of LLMs into EDA has opened new horizons for automating circuit design through advanced code generation techniques. Unlike digital design, which benefits from standardized components like logic gates and well-defined abstraction layers, analog design inherently deals with continuous signal processing, non-linear behaviors, and highly sensitive performance requirements. In analog circuits, even small changes in component values or topologies can result in significant variations in performance metrics, such as gain, bandwidth, or power consumption. This makes analog design less amenable to traditional automation techniques, as it requires precise modeling of physical components (e.g., transistors, resistors, and capacitors) and their interactions in a real-world context. Furthermore, analog design often lacks the robust hierarchical abstraction layers seen in digital workflows, where higher-level functionality can often be decoupled from lower-level implementation details. Instead, analog design workflows require a tight integration between topology, parameter selection, and performance validation, which increases the complexity of automation.

These complexities give rise to distinct subfields within analog design automation, each addressing a specific stage of the design process. Specification-to-netlist translation focuses on converting high-level descriptions, often in natural language, into machine-readable circuit representations (e.g., SPICE netlists). Topology synthesis, on the other hand, explores or generates new circuit configurations, aiming to identify novel or optimized topologies that meet design objectives such as gain, noise, or power efficiency. Schematic-to-netlist and netlist-to-schematic transformations bridge the gap between human-readable graphical representations and machine-readable textual descriptions, enabling better integration between design, simulation, and debugging workflows. Finally, layout generation translates circuit designs into detailed physical layouts that meet fabrication constraints, a process that requires careful attention to parasitics, **design rule checking** (**DRC**), and **layout-versus-schematic** (**LVS**) validation. Each of these categories presents unique challenges. For example, spec-to-netlist translation must ensure functional correctness while interpreting ambiguous or incomplete specifications. Topology synthesis faces the difficulty of navigating vast and complex design spaces, often requiring the exploration of millions of potential configurations. Schematic-to-netlist and netlist-to-schematic transformations must maintain consistency between representations while ensuring that the design intent is preserved. Layout generation requires not only the accurate placement and routing of components but also the optimization of performance metrics such as area, power, and signal integrity while adhering to manufacturing constraints.

LLMs, with their ability to process and generate natural language, code, and structured data, have the potential to address these challenges by automating traditionally labor-intensive tasks and enabling faster design iterations. As progress continues, LLMs could serve as a critical tool for bridging the gap between high-level design intent and low-level implementation, transforming the historically manual and expertise-driven process of analog circuit design.

Spec-to-Netlist. AnalogCoder [102] is a training-free LLM agent designed for analog circuit design through Python code generation. The methodology includes a feedback-enhanced flow with tailored domain-specific prompts, enabling automated and self-correcting design of analog circuits. The LLM generates Python code for the desired analog circuit functionalities, leveraging its strong Python programming capabilities. A circuit tool library archives successful designs as reusable modular subcircuits, simplifying the creation of composite circuits. The feedback-enhanced design flow involves several stages, including requirement check, simulation and operating point check, DC sweep check, and function check, to ensure the correctness of the generated code. AnalogCoder outperforms existing LLM-based methods, such as GPT-4o, in solving circuit design tasks, successfully completing 20 out of 24 benchmark tasks, which include both basic and composite circuits.

Atelier [103] is a novel framework leveraging general-purpose LLMs and a curated domain-specific knowledge base to automate analog circuit design. Addressing challenges of data scarcity and domain expertise requirements, Atelier employs multiple specialized LLM-based agents structured in a **Graph-of-Thoughts** (**GoT**) architecture, with each agent responsible for specific design steps such as topology selection, circuit analysis, parameter tuning, topology modification, and design decision-making. The framework integrates advanced techniques like RAG, CoT reasoning, self-reflection, backtracking, and external tools (e.g., CMA-ES optimizer) to enhance interpretability and design efficiency. Experimental results demonstrate Atelier's superiority over black-box optimization methods, general-purpose LLMs, and state-of-the-art LLM-based methods like Artisan, achieving higher success rates, superior performance metrics, and reduced runtimes across diverse opamp design tasks.

SPICECopilot [104] introduces a framework designed to automate SPICE code generation and simulation using LLMs and the PySpice Python library. SPICEPilot addresses the limitations of existing LLMs in hardware-specific tasks, such as inaccurate gate width/length ratios, improper input signals, and incorrect circuit analysis. The framework generates SPICE code through prompt engineering, validates it using automated feedback loops, and creates a comprehensive open-source dataset to mitigate data scarcity in analog circuit design. It also introduces benchmarking metrics for evaluating LLM performance across various circuit complexities and demonstrates superior results compared to existing analog circuit generation benchmarks.

PICBench [105] introduces the first comprehensive open-source benchmarking framework designed to evaluate the capabilities of LLMs in automating **Photonic Integrated Circuits** (**PICs**) design. PICBench includes 24 meticulously crafted design problems, spanning optical computing, optical interconnects, optical switches, and fundamental devices, with varying complexities and scales. The framework enables automatic evaluation of LLM-generated PIC netlists through syntax validation and functionality testing using the open-source simulator SAX. A feedback-based prompt engineering method is proposed to iteratively refine designs, addressing common syntax and functional errors, thus enhancing model performance. The study evaluates state-of-the-art LLMs, such as GPT-4 and Gemini 1.5 Pro, demonstrating the impact of feedback mechanisms and restrictions in improving syntax and functionality correctness.

Topology Synthesis. Artisan [106] is an automated operational amplifier (opamp) design framework that leverages a domain-specific LLM to integrate knowledge-driven reasoning with code generation for analog circuits. It introduces a bidirectional circuit representation that aligns netlists with natural language descriptions, enabling semantic understanding of circuit topology and functionality. Using **Tree-of-Thoughts** (**ToT**) and CoTs methodologies, Artisan models the hierarchical opamp design process as a multi-agent QA sequence, facilitating tasks like topology selection, parameter tuning, and performance optimization. Artisan generates behavioral and transistor-level netlists, automating the analog design process while offering interpretability and efficiency. Experimental results demonstrate that Artisan outperforms state-of-the-art black-box optimization methods (e.g., Bayesian and RL approaches) and general-purpose LLMs by achieving higher success rates, competitive circuit performance, and up to 50.1× faster design cycles.

LaMAGIC [107] leverages **supervised fine-tuning** (**SFT**) of LLMs for automated topology generation of analog circuits, specifically power converters. The framework enables direct one-shot generation of optimized circuit designs from user-defined specifications, overcoming the inefficiencies of traditional manual design and simulation-intensive search methodologies like RL. LaMAGIC introduces and evaluates multiple innovative circuit formulations, including canonical, adjacency-matrix-based, and float-input-based representations, to address challenges in representing analog circuits as graphs and ensure compatibility with the autoregressive nature of LLMs.

Experimental results demonstrate LaMAGIC's success rate of up to 96% under a strict tolerance of 0.01 for both edge and topology generation tasks, with adjacency-matrix formulations showing superior scalability and adaptability to complex six-component circuits.

Schemetic-to-Netlist. Auto-SPICE [108] is a fully automated framework that leverages LLMs and computer vision techniques to generate SPICE netlists—a textual representation of analog circuits—from schematic images. The framework addresses key challenges in component recognition, net connectivity, and netlist verification by combining object detection (e.g., YOLO for component detection), deep Hough transform for net detection, and prompt tuning for LLMs like GPT-4. Auto-SPICE builds a pipeline to extract SPICE netlists directly from analog circuit diagrams in textbooks and research papers, reducing the need for manual annotations. The framework also includes a feedback-based netlist verification mechanism to improve accuracy. Using Auto-SPICE, the authors created a dataset of 2,100 SPICE netlists, which was used to fine-tune GPT-4-mini for improved netlist generation from circuit descriptions. Experimental results show that Auto-SPICE outperforms prior methods (e.g., AMSNet) in accuracy and scalability, achieving 100% structural similarity in generated netlists.

Netlist-to-Schemetic. Schemato [109] introduces a fine-tuned LLM designed to convert circuit netlists into human-readable schematics in two formats: LTSpice-compatible .asc files and CircuitTikZ-based LaTeX code for visualizing schematics. Schemato leverages supervised instruction tuning and data augmentation on a curated dataset of netlist-to-schematic pairs, enabling it to outperform state-of-the-art models, including GPT-4o and Llama-3.1-8B, across multiple evaluation metrics. It achieves a **compilation success rate** (**CSR**) of up to 93.75% for the netlist-to-LATEX task and generates schematics with a **mean structural similarity index measure** (**MSSIM**) three times higher than baseline models. Unlike heuristic-based approaches, Schemato generalizes across circuit types without requiring predefined templates or aesthetic rules by learning from human-designed examples. While demonstrating high accuracy for simpler circuits, the model struggles with larger, more complex topologies, highlighting the need for larger and more diverse training datasets.

Layout Generation. LayoutCopilot [110] is an interactive analog layout design framework powered by multi-agent collaboration. It accepts both high-level design intents and concrete requests in natural language from designers and then transforms them into precise executable commands. Concrete requests are directly handled by a *concrete request processor* that decomposes concrete tasks into subtasks corresponding to layout tool commands. For abstract requests, high-level suggestions are generated, which are transformed to concrete requests if they are accepted by the designer.

The GLayout framework [111] translates analog layout into a compact, technology-generic text representation. It fine-tunes an LLM using RAG to understand this text representation and generate layouts based on new information provided in-context. The LLM is trained on a dataset of less than 50 unique circuits and text documents providing layout knowledge. The framework uses open-source LLMs with varying parameter sizes (3, 7, and 22 billion parameters) to handle tasks ranging from simple 2-transistor circuits to more complex $\Delta\Sigma$ ADCs. The LLM is capable of translating general human language prompts into analog layout, making the design process more intuitive and accessible. Figure 7(b) illustrates GLayout overall flow.

3.6 Layout Applications

ChatPattern [112] is an LLM-powered layout pattern generation framework. The front-end LLM agent interprets user requirement, breaks down complex demands into manageable sub-tasks, which are executed by the layout pattern generation back-end model (a conditional discrete diffusion model). The back-end model supports the front-end agent by providing a set of fundamental APIs,

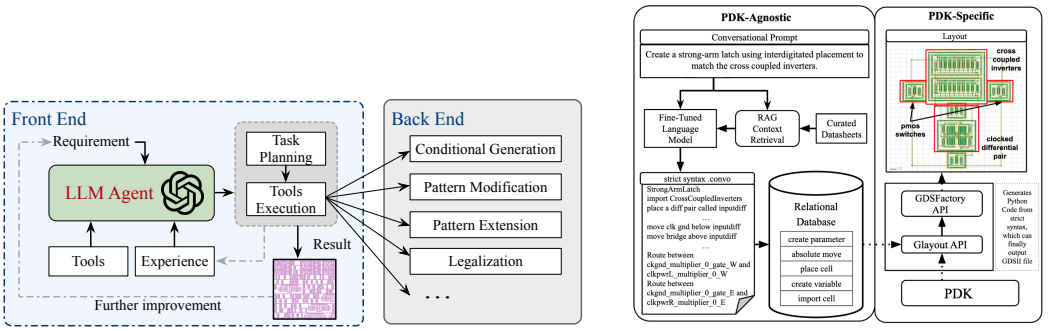


Fig. 7. LLMs enable automating tasks in back-end design stages. (a) Pattern generation by ChatPattern [112]; (b) Analog layout automation by GLayout [111].

such as random topology generation and topology legalization. Figure 7(a) illustrates ChatPattern overall flow. ChatPattern has been trained on 128×128 dataset for 250 GPU hours, and it outperforms in legality and diversity on both fixed-size and free-size pattern generation.

Chen et al. [113] introduce an intelligent **Optical Proximity Correction** (**OPC**) engineer assistant, which employs a two-stage methodology to automate the development of OPC recipes in semiconductor manufacturing. In the first stage, RL is used to explore and identify optimal solutions for OPC recipe generation, taking into account the shapes, positions, and surrounding contexts of patterns. The second stage leverages LLMs to synthesize and generalize the RL-generated results, converting them into practical OPC recipes. This involves data representation transformation, zero-shot feature generation, and the construction of a decision tree to facilitate the creation of optimized and efficient OPC recipes. The approach aims to reduce the time and expertise required for manual recipe adjustment while improving the overall effectiveness of the OPC process.

Francisco et al. [114] present an LLM-based Physical Verification Runset Generator, a generative AI solution leveraging Flan-T5, a fine-tuned LLM, to automate the creation of **physical verification** (**PV**) runsets for advanced semiconductor technology nodes. The system uses **design rule manual** (**DRM**) descriptions as input and generates PXL code (used by Synopsys IC Validator) to implement DRCs with up to 97% accuracy, significantly reducing the time and human effort required for runset creation. The solution incorporates a multi-stage workflow, including data pre-processing to standardize and decompose complex rules, model fine-tuning with curated DRM-PXL tuples, and post-processing to ensure syntax and semantic correctness of generated code. A **knowledge graph** (**KG**) is employed to track layer dependencies and enhance rule decomposition. Experimental results on FreePDK3 and commercial technology nodes demonstrate high accuracy in generating syntactically correct and executable PXL code. The framework also introduces a confidence scoring system to prioritize human review for low-confidence outputs.

DRC-Coder [115] leverages LLMs and **Vision-Language Models** (**VLMs**) to interpret multi-modal input, including textual rule descriptions, visual illustrations, and grid-based layout representations, for generating highly accurate DRC code. The framework divides the task into two hierarchical sub-tasks—rule interpretation and code generation—handled by two specialized LLM agents (Planner and Programmer). DRC-Coder incorporates domain-specific utility functions, including automated debugging and evaluation, to ensure alignment with commercial DRC tool outputs. Experimental results demonstrate that DRC-Coder achieves perfect F1 scores (1.000) across

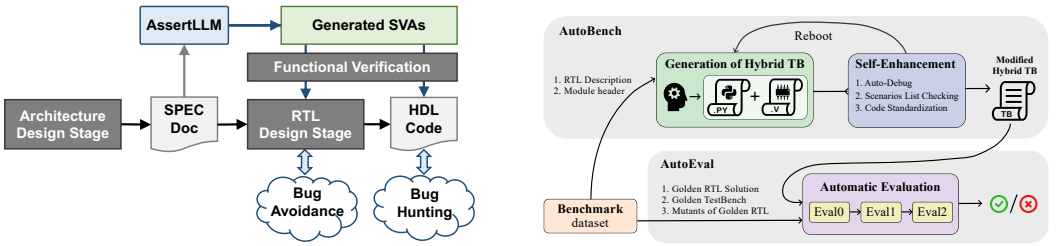


Fig. 8. LLM for hardware verification. (a) Assertion generation by AssertLLM [119]; (b) Testbench generation by AutoBench [120].

seven design rules, outperforming standard prompting methods (F1 = 0.631) and drastically reducing coding time from days to an average of four minutes per rule.

4 Verification and Debugging

LLMs for hardware verification present a cutting-edge approach to addressing the challenges of verifying complex hardware designs. Hardware verification, which ensures that a design behaves as intended, is a time-consuming and resource-intensive process, often involving tasks such as writing **SystemVerilog Assertions** (**SVA**), developing simulation-based testbenches, and addressing security vulnerabilities. LLMs have the potential to automate and enhance these critical aspects by leveraging their ability to interpret natural language specifications, learn from existing verification patterns, and generate high-quality, synthesizable code. For SVA generation, LLMs can translate functional requirements into precise assertions to ensure critical properties hold throughout the design. In testbench generation, they can produce **Universal Verification Methodology** (**UVM**) components, test stimuli, and coverage models to rigorously validate hardware behavior. Additionally, LLMs can assist in identifying and mitigating security vulnerabilities, such as side-channel attacks or hardware Trojans, by analyzing design specifications and suggesting verification strategies. By streamlining these processes, LLMs offer the promise of reducing verification effort, improving coverage, and strengthening design security, making them an indispensable tool in modern hardware development.

4.1 Assertion Generation

NL2SVA [116] translates the circuit under verification and the specifications in natural langauge into an array of intermediate representations, which contains candidate SVA. *Sub-translation* is also provided, which decomposes the requirement and maps formalization back to parts of the natural language. Designers can edit the sub-translation to incorporate human feedback.

Hassan et al. [117] present a methodology that combines LLMs with formal verification and mutation testing to generate and evaluate invariants for hardware designs. The approach is divided into three stages: generating invariants from design specifications, creating a formal model from the Verilog behavioral model, and performing a qualitative analysis of the invariants using mutation testing. This method leverages LLMs to automate the generation of invariants and formal models, while mutation testing is employed to assess the quality and robustness of these invariants.

Aditi et al. [118] introduce a method for automatically converting natural language specifications into validated SVAs using fine-tuned LLMs. The approach involves generating multiple candidate SVAs from the natural language specification, followed by a validation process that uses a second

language model to convert the generated SVA back into an English statement. This English statement is then compared to the original specification to ensure syntactic and semantic compatibility, thereby validating the generated SVA. Experimental results show that 92% of the specs were correctly converted to the corresponding SVAs.

AssertLLM [119] is an automatic assertion generation framework for complete specification files. The process is decomposed into three steps: information extraction, signal name mapping, and SVA generation. LLMs are guided to verify signal bitwidth, connectivity, and functionality. A benchmark that contains both golden specification documents and golden RTL designs is released. Figure 8(a) illustrates AssertLLM workflow. Evaluation on a full design with 23 I/O signals shows that 89% of the generated assertions are both syntactically and functionally accurate.

ChIRRAG [121] is a framework that leverages LLMs to generate SVAs from natural language specifications of a design. It first transforms unstructured design specifications into a standardized JSON format. This structured input is then provided to the LLM, which generates an initial set of SVAs. The generated assertions are subsequently validated through simulation, and any errors, such as syntax or simulation failures, are identified. The LLM iteratively refines the assertions based on feedback from the simulation logs, ensuring that the final SVA suite is both syntactically and semantically correct. The process includes manual intervention when necessary, particularly for implementation bugs or issues that the LLM cannot resolve automatically. Only 27% of the generated SVA require refinement upon iterative prompting.

LAAG-RV [122] presents a framework that also targets generating SVA from natural language specifications of RTL designs using LLM. It starts with preprocessing the RTL design files, dividing them into submodules, and extracting concise abstract descriptions that capture relevant design data such as I/O ports and functionality. These abstracts, along with the natural language specifications, are then fed into an LLM to generate initial SVA assertions. The generated assertions are validated through simulation, and any issues are iteratively refined by providing the LLM with carefully crafted manual prompts derived from simulation failures. This iterative process ensures that the final assertions are both syntactically and semantically correct, and a one-time Verilog loop is used to synchronize signal names, enhancing the quality of the generated SVAs.

Orenes et al. [123] explore whether LLMs can generate SVA for a given design without any specification beyond the RTL. Human efforts are involved in the iterative generation process to 'teach' LLMs how to generate correct assertions. This is done by crafting a rule set in plain natural language based on expert knowledge of SVA, such as "&bitarray means that ALL the bits are ONES". Their AutoSVA2 framework is able to generate correct SVA from buggy RTL, which improves coverage of RTL behavior by up to 6×, and has exposed a bug in the RISC-V CVA6 Ariane core.

4.2 Testbench Generation

AutoBench [120] is an LLM-automated testbench generation and evaluation framework for hardware verification. Given RTL description and module header, AutoBench first generates an HDL implementation, which is used to discriminate the circuit type into *CMB* or *SEQ*. A *Verilog driver track* and a *Python checker track* are responsible for generating the test stimuli input and the expected output, respectively. *AutoEval* introduced in the same article is a system to assess the generated test benches with three criteria, including syntactic correctness, no error with golden RTL, and a mutant-based coverage-oriented evaluation. Figure 8(b) illustrates AutoBench workflow. AutoBench demonstrates a 57% improvement in the testbench Eval2 pass@1 ratio, and 3.36 times Eval2 pass@1 ratio for sequential circuits. CorrectBench [124] is an extension of AutoBench, designed to overcome its key limitations in testbench generation for HDL design. While AutoBench relies solely on syntax self-checking and lacked functional validation and correction mechanisms, it leads to low success rates, particularly for sequential circuits. CorrectBench addresses these shortcomings by

incorporating a scenario-based validator to perform functional self-validation using imperfect RTL designs and an LLM-driven corrector that utilizes bug information from the validator to iteratively fix errors in generated testbenches. This extension enhances performance, achieving a 70.13% pass ratio compared to AutoBench's 52.18%, and demonstrates a 66.18% improvement for sequential circuits.

LLM4DV [125] presents a test stimuli generation framework. A *coverage plan*, i.e., a list of coverage points to be tested, is maintained and provided to the LLM at each iteration step, where LLM is requested for a test stimuli. Sampling techniques are utilized to handle the large number of uncovered *bins*.

Bhandari et al. [126] discuss LLM-aided Testbench Generation for **Finite-State Machines (FSMs)**. Given the specification and the design RTL of DUT, LLM is prompted to generate an initial testbench, which undergoes a compilation check using commercial tool. If any errors are detected during compilation, these errors are appended to the prompt and fed back to the LLM for refining the testbench. Next, the FSM is simulated in coverage analysis mode, where uncovered FSM transitions are reported and fed back to LLM to generate new test cases.

LLM-TG [127] is a framework leveraging LLMs to automate and enhance processor test case generation for design verification. LLM-TG addresses challenges in traditional verification, such as the time-intensive and error-prone nature of manual test case creation, by employing a structured workflow comprising prompt-based test case construction, LLM-assisted optimization, test result feedback, and a reusable prompt library. The framework generates both inline assembly and C programs, improves test case correctness through iterative feedback, and boosts test efficiency by reusing validated prompts. Experiments on an RV64GC-based processor (Xuantie-C910) demonstrate that LLM-TG achieves 8.34% higher block coverage and 5.8% higher expression coverage compared to state-of-the-art methods like RISCV-DV and LLM4DV. Additionally, the prompt library reduces token consumption by over 50%, accelerating subsequent verification processes.

VerilogReader [128] introduces a framework that integrates LLMs, into the **Coverage Directed Test Generation (CDG)** process for automatic hardware test generation. The framework positions LLMs as "VerilogReaders", enabling them to comprehend Verilog code and generate test stimuli to maximize code coverage. Key innovations include the DUT Explainer, which provides natural language descriptions and test guidance for designs, and the Coverage Explainer, which reformats coverage reports into LLM-readable formats to assist in generating inputs for unexplored code branches. Evaluated on a benchmark suite of 24 Verilog designs, VerilogReader outperforms random testing, achieving full code coverage on simple and medium-level designs while highlighting LLM limitations with more complex circuits. The study also explores scalability challenges, showing a decline in performance with larger designs, and suggests future directions such as integrating LLMs with **Graph Neural Networks (GNNs)** to improve structural comprehension and scalability for real-world hardware verification tasks.

4.3 Security

DIVAS [129] identifies security vulnerabilities from user-defined SoC specifications. It first maps them to the relevant CWEs, followed by the generation of equivalent assertions. An extensive CWE database is constructed to help filter LLM response.

Kande et al. [130] investigate the use of LLMs for hardware security assertion generation. To enable this, designer intents are captured in a comment string specifying the properties that are checked. Design source, example assertions, together with the comment strings, are organized into a prompt string to query the LLM. Out-of-the-box code LLMs are used to comprehensively evaluate the framework, achieving an accuracy of up to 93.55% across all the benchmarks, compared to the 26.54% average accuracy with all prompt configurations of varying levels of context.

SPICED [131] introduces a framework designed to detect and localize syntactical bugs and analog Trojans in circuit netlists. The methodology employs CoT reasoning and few-shot examples to teach the LLM anomaly detection rules. Specifically, the LLM is provided with SPICE netlists and corresponding simulation log files, along with detailed circuit specifications. The LLM is trained to identify and flag syntactical bugs such as missing nodes, incorrect component definitions, and floating nodes, and to detect and locate Trojans by analyzing deviations in voltage and current behaviors. The process involves iteratively refining the LLM's understanding of SPICE syntax and circuit behavior through feedback, ensuring that it can accurately pinpoint and describe the nature and location of detected anomalies. This approach leverages the LLM's NLP capabilities to enhance the detection and localization of both syntactical bugs and Trojans without requiring explicit training or hardware modifications. It achieves an average Trojan coverage of 93.32% and an average true positive rate of 93.4% in identifying Trojan-impacted nodes for the evaluated analog benchmark circuits.

LASP [132] (LLM Assisted Security Property Generation) is a framework designed to automate the generation of security properties for SoC verification. The methodology begins with preprocessing and decomposing the RTL design into sub-modules, extracting concise summaries that include module hierarchy, interfaces, and key design features. These summaries, along with the design specifications and a detailed threat model, are then provided to an LLM. The LLM generates natural language security properties based on this information. The process also involves a hybrid approach that combines static analysis to extract control and data flow features with LLM capabilities to identify additional security-critical features, ensuring comprehensive coverage of security assets. 80% of the generated properties were readily translatable into SVAs and tested using formal tools.

Assert-O [133] extracts security properties from hardware documentation, merges similar properties, and prioritizes them based on their criticality to system security. In particular, it merges overlapping properties that share operations or register types. Using the **Common Vulnerability Scoring System (CVSS)**, it ranks security properties by their severity and assigns weights based on the role of the corresponding IP in the system (e.g., cryptographic cores and peripherals). Assert-O was trained and evaluated on OpenTitan, an open-source SoC design, where it processed documentation for six IPs, initially generating 183 security properties, which were optimized and reduced by 24.6% to 138 properties. The properties were ranked based on their severity, with the *Reset Manager* identified as the most security-critical module.

SoCureLLM [134] is a hardware security verification framework that handles large-scale SoC designs. Its automated vulnerability detection method partitions the design into code snippets, generates additional context through summarization, and detects security policy violation. New security policies are created through threat model-based assessments. It detected 76.47% of security bugs across three vulnerable RISC-V SoCs, and develops 84 new security policies from additional designs.

4.4 Debugging

Traditional hardware design debugging often involves meticulously reviewing lines of code, manually identifying syntax errors, or logic flaws, which can be time-consuming and error-prone. LLMs, with their deep understanding of programming languages and hardware design principles, can analyze and interpret complex HDLs like Verilog or VHDL. They offer intelligent suggestions for code optimization, identify potential bug patterns, and even autonomously correct errors, thereby expediting the debugging cycle.

RTLFixer [135] is a debugging framework that utilizes LLMs as autonomous language agents enhanced with RAG and ReAct. In particular, an LLM is designed for code generation. RAG incorporates human expert guidance (e.g., how to solve a common error). The ReAct prompting

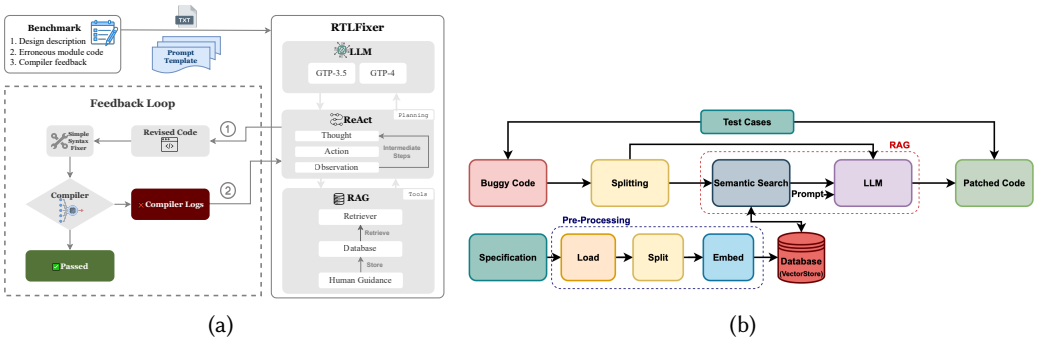


Fig. 9. Debugging frameworks of (a) RTLFixer [135] and (b) Qayyum et al. [142]. Both works leverage RAG techniques, for human guidance and specification retrieval, respectively.

mechanism [136] handles the iterative code refinement with intermediate reasoning and action steps. In ReAct, the LLM prompts itself for thoughts on how to address the error and selects the next action, which could be generating an explanation for the error, searching for a solution in the human expert guidance database, revising the code, and submitting the revised code to the compiler. Figure 9(a) illustrates the debugging framework of RTLFixer. The method leads to a 32.3% and 10.1% increase in pass@1 success rates in the VerilogEval-Machine and VerilogEval-Human benchmarks, respectively.

MEIC [137] introduces an iterative debugging framework, at the heart of which is the *Debug Agent*, an LLM specifically fine-tuned to identify and correct both syntax and functional errors in RTL code. The Debug Agent acts as an intelligent intermediary, receiving inputs such as design specifications, untested RTL code, compilation logs, and simulation logs. Leveraging its training on a rich dataset of common RTL errors, it parses through the code, comprehending the intended functionality and diagnosing issues. It then produces a refined version of the RTL code, aiming to correct the identified errors. This transformation is a direct code-to-code operation where the raw, potentially faulty RTL code is transformed into a corrected variant through the LLM's generative abilities. It achieves fix rate of 93% for syntax errors and 78% for function errors, with up to 48× speedup in debugging processes when compared with experienced engineers.

Ahmad et al. [138] quantitatively evaluate the performance of LLM repairing hardware security bugs. Input messages to the *repair generator* comprises the fault location, bug description, and fix instructions. The framework notably supports DSE of prompts (i.e., prompt engineering) and identifying the best parameters for the LLM. They observe that fixes that span more than 1 line or require the removal of a buggy line are harder to repair.

LLM4SecHW [139] proposes to finetune LLM for harware debugging utilizing version control data. Downstream tasks include bug localization (code removed during repair are considered defective) and repair.

Self-HWDebug [140] leverages the self-instructional capabilities of LLMs to produce debugging instructions autonomously. For a specific vulnerability, for example a CWE, debugging instructions are generated by prompting the LLMs with a predefined pair of vulnerable and secure code snippets, known as the reference sample. The generated security rule checks are then used to debug new vulnerable code.

HDL-Debugger [141] proposes a comprehensive solution for HDL debugging, consisting of HDL debugging data generation via a reverse engineering approach, a search engine for RAG, and a retrieval-augmented LLM fine-tuning approach. For data generation, it creates buggy HDL code

by applying modification functions to correct code based on known error patterns. Modification functions are created by leveraging LLMs to analyze HDL manuals, expert notes, and user logs, extracting prevalent error patterns. An array of high-quality, industry-provided HDL codes serve as the basis for applying the generated modification functions, thereby producing a dataset of code pairs encompassing both faulty and corrected versions. HDL-Debugger achieves a pass-rate of up to 81.93% on an HDL dataset sourced from Huawei.

Qayyum et al. [142] present a method that combines RAG with LLM for HDL bug identification and patching. They first split the specification file into smaller chunks and store them in a vector database. Then, HDL code is split into single lines, which are used to query the vector database for matching specification chunks. In case the LLM identifies a bug in the HDL code w.r.t. the given semantically retrieved specification, it creates a patch following the same HDL code pattern. Figure 9(a) illustrates the debugging framework of Qayyum et al.

LiK [143] is an LLM-based solution for functional bug localization in Verilog. The model is fine-tuned on a bug location dataset, which is constructed by bug injection and reasoning thought reverse-engineering. LiK achieves a pass@1 localization accuracy of 93.3% on a test dataset based on RTLLM, surpassing GPT-4's 77.9% and comparable to Claude-3.5's 90.8%. The bug location obtained by LiK improves GPT-3.5's bug repair efficiency (functional pass@1) from 40.39% to 58.92%.

Wan et al. [144] construct the Chrysalis dataset by injecting logic bugs into HLS designs. Over 1,000 individual HLS programs are first collected from open-source projects. Then, common pre-silicon logic bugs are listed, including OOB access, infinite loop, and so on. Erroneous code are generated by injecting one or two types of bugs into the original design, which is done by prompting LLMs with context, requirement, and some instructing rules.

SPICED+ [145] is an enhanced extension of the SPICED framework, specifically designed to not only detect and localize analog Trojans in A/MS (Analog/Mixed-Signal) circuits but also to mitigate them through iterative correction. While SPICED focuses on identifying Trojan-impacted nodes and detecting anomalies using LLMs guided by supervised learning rules and few-shot examples, SPICED+ introduces a confidence scoring metric and integrates multiple tools, including SPICED and HSPICE, to refine the detection process and ensure precise mitigation. SPICED+ iteratively removes suspect Trojan lines from SPICE netlists, reruns HSPICE simulations to validate circuit performance, and employs state management for rollback in case of adverse impacts. By incorporating confidence scores to prioritize suspect nodes and automating the correction workflow, SPICED+ achieves an average Trojan coverage of 93.3%, a **Trojan mitigation rate** (**TMR**) of 94.6%, and a false-positive rate of just 1.4%, significantly improving upon SPICED's capabilities while maintaining zero area and power overheads.

5 Knowledge Representation and Retrieval

LLMs for knowledge representation and retrieval are redefining how information is organized, accessed, and utilized in EDA. As hardware designs grow increasingly complex, the ability to effectively represent and retrieve knowledge from design specifications, technical documentation, and debugging logs has become critical. LLMs, particularly those fine-tuned for EDA-specific tasks, excel in extracting, structuring, and reasoning over vast amounts of unstructured and semi-structured data. They enable context-aware QA, assist in reviewing specifications, and identify patterns that aid in bug detection and analysis. By bridging the gap between natural language and technical design data, LLMs offer a transformative approach to streamlining EDA workflows, reducing manual effort, and enhancing the accuracy and efficiency of design and validation processes. Their ability to dynamically adapt to domain-specific knowledge makes them invaluable in addressing the growing information complexity in modern EDA systems.

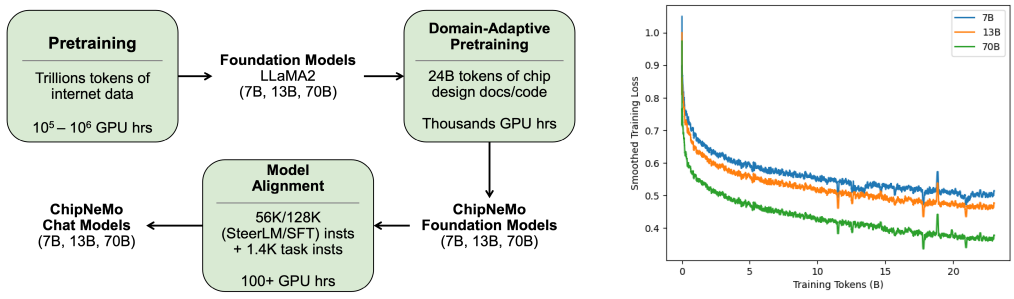


Fig. 10. Training of the ChipNeMo model [91]. (a) Training flow; (b) Training loss.

5.1 Domain Foundation Models

The ChipNeMo model [91] is a domain-adapted LLM for chip design, built by starting with a pre-trained LLaMA2 model. It undergoes **domain-adaptive pretraining** (**DAPT**) using a large corpus of proprietary hardware-related code and natural language datasets, such as RTL, verification testbenches, and documentation. The tokenizer is also adapted to include new tokens specific to the chip design domain, enhancing its ability to handle specialized terminology. The model is further aligned with domain-specific tasks through fine-tuning on a combination of general and domain-specific instruction datasets, ensuring it can effectively address chip design-related queries and tasks. Figure 10 illustrates the training flow of the ChipNeMo model.

Hardware Phi-1.5B [146] is an LLM specifically tailored for the hardware domain of the semiconductor industry. The methodology involves developing a specialized, tiered dataset comprising small, medium, and large subsets, with a focus on pre-training the model using the medium-sized dataset. This approach leverages the compact yet efficient architecture of the Phi-1.5B model [147]. Additionally, the model undergoes DAPT to encode hardware-specific knowledge, enhancing its capabilities in tasks related to hardware design and verification. The training process is designed to address the complexity of hardware-specific issues that are not adequately covered by general natural language or software code knowledge.

5.2 Specification Generation and Review

SpecLLM [148] explores the application of LLMs in architecture specification generation and review. They categorize the architecture specifications into three levels: high-level (e.g., the RISC-V spec), middle-level (e.g., a RISC-V CPU spec with an overview of microarchitectures), and low-level (e.g., a spec with implementation details for each microarchitecture). Accordingly, they collect a dataset of specifications of various levels, organized by product types like CPU, SoC, Accelerator, Bus and Network, Arithmetic and Crypto. Experiments indicate LLMs are capable of efficiently generating architecture specifications as well as providing valuable feedback to improve document accuracy.

5.3 Question-Answering

SmartonAI [149] is a multi-agent interaction assistant plugin for EDA software. It functions through two major components: the Main-Sub GPT and the QA GPT. The main GPT accepts user query and identifies relevant sub-tasks from a pre-classified set. Sub GPTs further interact with the user

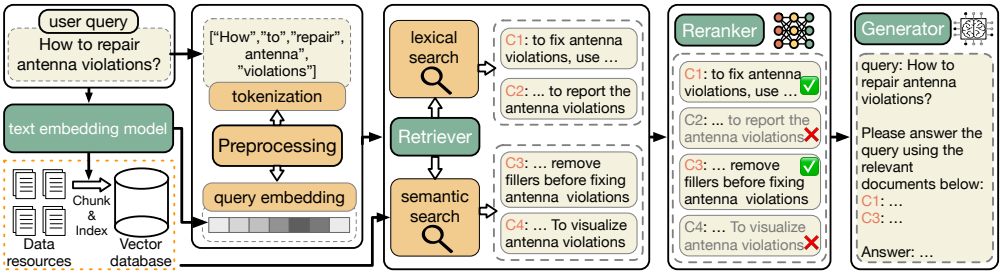


Fig. 11. Overview of RAG-EDA [150].

and generate personalized learning documents to the QA GPT. The QA GPT leverages the learning documents to provide responses to the user's initial queries.

RAG-EDA [150, 151] is a customized RAG framework tailored for EDA tool documentation QA. It enhances the text embedding, reranker, and generator models with domain-specific techniques, including contrastive learning for fine-tuning, distillation from a proprietary LLM, and instruction tuning on an EDA-related corpus. Additionally, the authors developed and released ORD-QA, a benchmark for evaluating QA systems based on OpenROAD documentation, which consists of 90 high-quality question-document-answer triplets. The proposed Crag-EDA outperforms state-of-the-art methods on both the ORD-QA and a commercial EDA tool's documentation QA benchmark. Figure 11 illustrates the overall flow of RAG-EDA.

ORAssistant [152] is a RAG-based conversational assistant designed to enhance the user experience for the OpenROAD flow. Built on a scalable, tool-based RAG architecture, ORAssistant leverages curated datasets from OpenROAD documentation, GitHub discussions, and other public resources to provide context-specific and reliable responses to user queries related to installation, command usage, flow setup, and debugging. By integrating hybrid retrieval techniques, including semantic and keyword-based searches, ORAssistant ensures accurate and context-aware answers with citations and hyperlinks. Evaluation results demonstrate that ORAssistant significantly outperforms base pre-trained LLMs (e.g., GPT-4o and Gemini 1.5 Flash) in accuracy, precision, and relevance across OpenROAD-related QA tasks, achieving F1 scores exceeding 93% on custom datasets.

EDACopilot [153] is a RAG-powered intelligent assistant designed to enhance LLM performance across the RTL-to-GDSII EDA flow. The proposed EDA-Adaptive RAG framework addresses challenges in processing EDA-specific unstructured data by integrating a query classification system and enhanced retrieval mechanisms, enabling precise and context-aware responses for diverse tasks such as terminology explanation, script retrieval, script annotation, and knowledge advice. A comprehensive EDA corpus was constructed to support this framework, incorporating structured datasets and verified RTL code samples. The framework also demonstrates its versatility through an application in RTL code generation, where it improves syntactic correctness and functional accuracy by leveraging relevant retrievals as input context. Experimental evaluations show that EDA-Adaptive RAG outperforms state-of-the-art models on EDA Q&A tasks and RTL code generation, achieving up to a 16.11% improvement for complex queries and a 36% increase in RTL functionality correctness compared to baseline models.

EDA-Corpus [154] is an open-source dataset tailored for OpenROAD, featuring both QA data pairs and prompt-script data pairs. The question-answer data are collected from OpenROAD GitHub issues, discussions, and documentation. The datapoints are categorized into three categories, namely, *general*, *tool*, and *flow*.

HDLCopilot [155] is an LLM-powered PDK query system. Given a natural language query, an SQL query is dynamically generated to retrieve information from the PDK database. Three views

of PDK are stored: *liberty* for storing timing information at different operating conditions, *LEF* for abstract layout information for each cell, and *Technology LEF* for storing metal stack properties. An *interpreter* agent then translates raw database result to a coherent natural language response.

Rittikulsittichai et al. [156] introduce an intelligent chatbot assistant designed to enhance PCB manufacturing through effective troubleshooting and knowledge dissemination, particularly for the electrolytic copper plating process. Built using the Langchain framework and leveraging RAG, the chatbot integrates ChatGPT and Llama 2 to deliver accurate, context-aware responses. A novel dataset of 602 question-answer pairs was constructed to serve as the knowledge base, and the chatbot's performance was evaluated using quantitative metrics (faithfulness, answer relevancy, context recall, and precision) and qualitative assessments by domain experts. While quantitative results showed no significant differences between models, qualitative feedback favored the ChatGPT-based chatbot (TEA/CGT) for its superior naturalness, coherence, and informativeness. The study highlights the chatbot's potential to reduce operational delays, improve efficiency, and set new performance standards in PCB production. Future directions include enhancing capabilities for complex calculations, image-based query understanding, and automating knowledge base creation to further streamline industrial applications.

Ask-EDA [157] introduces a conversational agent designed to assist electronic design engineers by leveraging LLMs, hybrid RAG, and **Abbreviation De-hallucination** (**ADH**) techniques. Ask-EDA uses LLMs to generate natural language responses, RAG to enhance these responses with information retrieved from an external knowledge base using both dense and sparse retrieval methods, and ADH to ensure accurate expansion of abbreviations by consulting a pre-built abbreviation dictionary. The system is tailored to address various tasks in design construction, verification, and technology development, providing more relevant and accurate guidance to design engineers.

ChipExpert [158] is an open-source instructional LLM tailored for the IC design field. The training process of ChipExpert includes continue pretraining, supervised fine-tuning on domain-specific QA pairs, and alignment through DPO. RAG is utilized during inference to retrieve top 3 similar passages for reference. The QA benchmark ChatICD-Bench is also released.

ChipAlign [159] presents a training-free approach to improve instruction alignment in domain-adapted LLMs for chip design. Current chip-specific LLMs, such as ChipNeMo, often lose general-purpose instruction-following capabilities during domain adaptation, limiting their usability in practical applications like QA and multi-turn interactions for hardware engineers. To address this, ChipAlign employs geodesic interpolation on a Riemannian manifold to merge the weights of a chip-specific LLM with a general instruction-aligned LLM, effectively combining chip expertise with strong instruction-following abilities. The method demonstrates superior performance across benchmarks, achieving a 26.6% improvement on the IFEval instruction-following benchmark and significant gains on chip-specific QA tasks, including 3.9% on OpenROAD QA and 8.25% on industrial chip QA.

Fernando et al. [160] propose a documentation generation framework, which takes specification as input, and generates design documentation by prompting LLMs. Following a three-layered workflow, the *specification* is first transformed into *documentation*, which are further printed to desired formats like LaTeX in the *view* layer. LLMs could be prompted by either manually written sentences or simply a few keywords (e.g., "Here ADD is: addition, no carry, 32 bit").

Wang [161] discusses AI-assisted test data analytics. In particular, domain grounding and KG are introduced to constrain LLM response for domain-specific contexts.

CIRCUIT [162] is a specialized benchmark designed to evaluate the reasoning and interpretation abilities of LLMs in the domain of analog circuit design. The dataset consists of 510 question-answer pairs derived from 102 templates across four categories (Basic, Analog, Power, and RF circuits) and three complexity levels, with associated diagrams, netlists, and varying numerical setups. CIRCUIT

employs a unit-test-like evaluation metric, pass@k/n, to assess an LLM's ability to understand circuit topologies and numerical relationships, providing deeper insights into performance beyond global accuracy metrics. Experiments with leading LLMs such as GPT-4o, GPT-4-turbo, and Gemini 1.5-pro reveal significant limitations in reasoning about circuits, particularly with topologies involving netlists, as the best-performing model (GPT-4o) achieves only 48.04% global accuracy and 27.45% pass@5/5 template accuracy. The study highlights reasoning errors, such as topology misunderstandings and direction errors, as key challenges for LLMs in this domain.

AMSnet-KG [163] is a comprehensive annotated dataset designed to facilitate large-scale **analog and mixed-signal** (**AMS**) circuit automation using LLMs and KGs. AMSnet-KG includes 894 circuits with schematics, netlists, and annotations structured into a KG for efficient retrieval and usage. Leveraging this dataset, the authors propose AMSgen, a data-driven framework for automatic AMS circuit generation. AMSgen integrates LLM-generated design strategies, RAG from AMSnet-KG, and **Bayesian optimization** (**BO**) for topology selection and transistor sizing. The framework iteratively refines circuit designs to meet performance specifications with minimal human intervention. Experiments on operational amplifiers and comparators demonstrate the effectiveness of AMSgen in generating fully sized, high-performance netlists under 28 nm technology.

DocEDA [164] automates the extraction of electrical parameters and circuit design optimization from datasheets and technical documents. The framework integrates computer vision techniques with LLMs and includes five key components: a specialized layout analysis model (EDocNet) for document segmentation, parameter extraction using CoT reasoning, a structured database for storing extracted circuit information, an image-to-circuit transformation process using the GAM-YOLO model for topology and netlist generation, and a dynamic optimization framework for refining circuit layouts with space mapping and BO. The framework achieves over 90% accuracy in image-to-circuit transformations and demonstrates superior performance in optimizing analog circuit designs, such as low-noise amplifiers.

5.4 Bug Analysis

ChipNeMo [91] introduces an application of bug summarization and analysis. The methodology for this application involves several key domain adaptation techniques, including adapted tokenizer for specialized vocabulary, DAPT using in-domain data, and domain-specific instruction tuning. As for bug summarization and analysis, the approach includes hierarchical summarization, where the LLM processes large and complex bug reports by breaking them down into manageable chunks, summarizing each, and then combining these summaries into a coherent final summary. This process is designed to handle the long comment histories and code snippets often found in bug reports, making the information more accessible and actionable for engineering teams.

Qiu et al. [165] investigate if LLMs can be used to generate novice-friendly explanation of synthesis errors. A corpus of representative synthesis-time bugs (but not run-time issues) is established by extracting from synthesizer (Vivado, Quartus) error log files. LLMs are prompted for pedagogically-focused error message explanations, which are manually graded over a set of metrics.

6 Optimization and Modeling

Optimization and modeling leverage the advanced reasoning and generative capabilities of LLMs to enhance traditional EDA workflows. These tasks involve using LLMs to assist in refining circuit parameters, exploring design tradeoffs, and generating efficient models for predicting system behavior. For example, LLMs can help optimize PPA metrics by analyzing design constraints, generating optimization strategies, or automating iterative processes. Additionally, they can aid in creating high-level abstractions or surrogate models to represent complex systems, enabling

faster simulations and evaluations. By integrating LLMs into optimization and modeling workflows, engineers can benefit from improved automation, scalability, and decision-making in tackling the challenges of modern EDA.

LADAC [166] leverages a domain-specified LLM to assist in the design of analog circuits. The methodology involves equipping the LLM with a local knowledge library rich in domain-specific insights and interactive tools that allow it to autonomously size transistors and obtain simulation outcomes. The LLM is trained to formulate optimized design strategies based on user specifications and foundational principles of analog circuit design. This approach integrates an LLM as a decision-making agent, which can iteratively refine the design through a combination of NLP and domain-specific knowledge. Authors showcased designing a two-stage amplifier, a 3-stage NMCNR amplifier, and a ring oscillator with LADAC.

AmpAgent [167] is a multi-agent system based on LLMs for designing complex multi-stage amplifiers. AmpAgent consists of three agents: a Literature Analysis Agent, a Mathematics Reasoning Agent, and a Device Sizing Agent. The Literature Analysis Agent retrieves key information from literature, the Mathematics Reasoning Agent decomposes the design problem by deriving key formulas, and the Device Sizing Agent, integrated with simulation interfaces and conventional optimization algorithms, sizes the multi-stage amplifier according to the divided sub-problems. This collaborative approach enables efficient and systematic design of multi-stage amplifiers, with each agent contributing to different aspects of the design process. AmpAgent has improved circuit performance by up to 27.25×, while reducing number of iterations and execution time.

The LLANA framework [168] integrates LLMs with BO to enhance the generation of analog design-dependent parameter constraints. It leverages the few-shot learning abilities of LLMs to improve the efficiency of BO, enabling more effective exploration of the analog circuit design space. The LLMs are used to generate high-quality design points, while the BO process balances exploration and exploitation. The LLM is provided with natural language representations of BO components, allowing it to harness the unique advantages of LLMs, such as contextual understanding and rapid learning, to generate and refine analog layout constraints.

ADO-LLM [169] also combines LLM with BO, leveraging the LLM's ability to infuse domain knowledge to rapidly generate viable design points, addressing the inefficiency of BO in finding high-value design areas. The LLM is initialized with the circuit definition, design specifications, and design instructions, and it generates initial design points. These points are then evaluated using a circuit simulator, and the outcomes are used to further enrich the LLM's training examples, creating a feedback loop that improves the quality of the shared dataset and the effectiveness of the BO process. The LLM is also guided through a series of steps, including interpreting the circuit definition, balancing tradeoffs in design specifications, and providing few-shot examples with diverse simulation results to enhance its design capabilities.

LEDRO [170] (LLM-Enhanced Design Space Reduction and Optimization) is a framework that synergizes LLMs with optimization techniques to automate and enhance analog circuit sizing. LEDRO iteratively refines the high-dimensional design space for operational amplifiers (Op-Amps) by leveraging the reasoning capabilities of LLMs to identify relevant subspaces and employing optimization algorithms, such as TuRBO, to explore these regions efficiently. The framework addresses the limitations of traditional methods like RL and BO, which lack domain-specific insights and are costly to generalize across different circuit topologies and technology nodes. LEDRO demonstrates superior performance and generalizability across 22 diverse Op-Amp topologies spanning four FinFET technology nodes, achieving up to 48% improvement in the **Figure of Merit** (**FoM**) and up to 2.15× speed-up compared to state-of-the-art baselines.

RALAD [171] is a framework that uses retrieval-augmented LLMs to automate HLS code optimization. By dynamically retrieving relevant HLS examples from external knowledge bases, RALAD

guides LLMs to generate optimized code with minimal manual intervention. The framework includes several components: document embedding, retrieval of top-K relevant code snippets, prompt engineering (combining user queries with retrieved examples), and code generation. RALAD also uses manual annotations and hints in the prompts to improve the model's ability to insert HLS-specific pragmas (e.g., loop unrolling and array partitioning). Experiments show significant improvements, with up to 16× speedup in tasks like matrix multiplication and a 50% higher successful compilation rate.

PCBAgent [172] introduces an agent-based framework for high-density PCB placement that integrates an RL agent and an LLM agent to automate and optimize PCB layouts while adhering to industrial constraints. The RL agent utilizes a decision transformer to generate legal placements, optimizing objectives such as **half-perimeter wirelength** (**HPWL**) and **surface layer wires** (**NSLW**), while the LLM agent facilitates interactive, user-guided optimization by translating feedback into actionable commands. PCBAgent addresses challenges unique to PCB placement, such as irregular boundaries, high component density, and rotation feasibility, by incorporating novel techniques like placement sequence optimization, mask-based legalization, and HPWL-guided action search. Experimental results on 17 industrial PCB benchmarks demonstrate that PCBAgent achieves 90% of the quality of manual placements in HPWL and NSLW metrics and outperforms state-of-the-art methods.

GPCB routing [173, 174] applies **Generative Pre-Trained Transformers** (**GPTs**) to automate and optimize PCB routing. The approach transforms PCB routing into a network-flow problem, tokenizing routing patterns and leveraging GPT models for learning and predicting routing strategies based on human expertise. Key innovations include a sliding window mechanism, a local memory with cross-attention, and a multi-information fusion method that integrates flow encoding, start/end points, and capacity information to enhance prediction accuracy. GPCB enables parallel routing, significantly improving routability, runtime, and wirelength optimization compared to state-of-the-art methods. Experimental results demonstrate 100% routability, reduced signal latency through optimized wirelength, and up to 6.67× faster runtime than existing solutions like FreeRouting. By reimagining PCB routing as a token prediction task, GPCB showcases the potential of Transformer-based architectures for addressing complex, real-world routing challenges. Future directions include extending the method to bus routing, redistribution layers, and differential pair routing.

Xiao et al. [175] discuss leveraging LLMs for the synthesis and optimization of prefix circuits, which are critical components in digital adders for efficient carry signal computation. The framework, PrefixLLM, transforms prefix circuit synthesis into a structured text generation problem using a proprietary format called **Structured Prefix Circuit Representation** (**SPCR**). An iterative process is employed to generate valid SPCRs, ensuring correctness by integrating validation and feedback mechanisms. PrefixLLM also incorporates an iterative DSE framework to optimize prefix circuits for area and delay by utilizing a sorted circuit pool and custom prompts to guide the LLM in discovering high-quality designs. Experimental results demonstrate that PrefixLLM achieves up to a 3.70% area reduction compared to state-of-the-art methods and outperforms classical prefix circuits like Kogge-Stone and Brent-Kung under comparable delay constraints. The framework's efficacy is validated through both simulation and physical hardware implementation, including a chip tapeout process that shows significant reductions in area and routing wire length.

IICPilot [176] introduces an intelligent multi-agent framework designed to automate and optimize IC backend design processes using open-source EDA tools like OpenROAD and iEDA. Leveraging the capabilities of LLMs and a multi-agent system, IICPilot addresses challenges in backend design by automating tasks such as script generation, EDA tool invocation, DSE for performance optimization, and container-based resource allocation. The framework incorporates specialized

agents, including a DSE agent for optimizing PPA metrics (performance, power, and area) through tools like Autotuner and Hypermapper, a container agent for efficient resource allocation using Kubernetes, and a monitoring system to track and resolve workflow issues. Experimental results demonstrate improvements in automation, chip performance (up to 32.75% PPA improvement), and cost efficiency across various IC designs.

Ho et al. [177] introduce a methodology leveraging LLMs for optimizing standard cell layout design, specifically targeting PPA and routability in advanced technology nodes like 2nm. The proposed framework integrates LLMs with ReAct prompting to iteratively refine device clustering constraints, using designer expertise and netlist tools to dynamically improve cell layouts. The LLM comprehends SPICE netlists, physical layouts, and cluster constraint formats, enabling it to generate high-quality cluster constraints that address diffusion sharing, common gate connections, and routability issues. Experimental results demonstrate the effectiveness of the method, achieving up to 19.4% smaller cell area and generating 23.5% more LVS/DRC clean layouts compared to state-of-the-art baselines. The framework also achieves 100% success in fixing routability for complex sequential cells.

Modeling. LLSM [178] integrates LLMs and graph-based methods to enhance logic synthesis prediction. LLSM operates at the RTL level and aims to predict post-synthesis metrics like delay and area efficiently, bypassing the need for time-consuming synthesis tools. The proposed framework includes three key innovations: (1) EDA-guided CoT prompting, which uses step-by-step reasoning to extract circuit-level insights (e.g., gate numbers and critical paths) from RTL code; (2) Text-Circuit Hybrid Embedding, which combines LLM-extracted textual information with graph-based circuit embeddings to improve prediction accuracy; (3) AIG-tailored Acceleration Library, which introduces an optimized sparse data format and computation kernels for **And-Inverter Graph** (**AIG**) representations, accelerating graph processing tasks. LLSM achieves improvements in prediction accuracy, with reductions of 21.53% and 19.27% in delay and area prediction losses, respectively, compared to state-of-the-art methods. Additionally, LLSM demonstrates a 1.34× speedup over **PyTorch Geometric** (**PyG**) implementations and a 6, 296.77× speedup over traditional tools like Synopsys Design Compiler.

Amik et al. [179] present a novel ML framework for accurate and efficient timing prediction at the RTL in IC design. The framework addresses key limitations of existing ML-based methods, such as unreliable feature generation, lack of practical specifications (e.g., technology libraries and design constraints), and poor generalization across circuits. It consists of three components: an RTL dataset generator for creating diverse, large-scale datasets; a LLM-based RTL-to-graph parser for extracting node and graph-level features from technology files and design constraints; and a **Graph Attention Network** (**GAT**)-based ML engine for accurate timing slack prediction. The proposed method achieves an R^2 score of 0.98 in predicting **worst negative slack** (**WNS**) and **total negative slack** (**TNS**), surpassing state-of-the-art techniques. Furthermore, it demonstrates a 749× runtime speedup compared to commercial static timing analysis tools, enabling rapid DSE.

VeriDistill [180] is an end-to-end ML framework for predicting post-synthesis **quality-of-result** (**QoR**) metrics, such as area and delay, directly from raw Verilog code without preprocessing. The approach leverages the powerful representational capabilities of LLMs, specifically CodeV, to extract high-level semantic insights from Verilog code. These insights are further enhanced through a knowledge distillation process, where a GNN teacher model, trained on low-level **Look-Up Table** (**LUT**) graph representations, transfers circuit-level knowledge to the LLM-based predictor. During training, a regularization term aligns the LLM's representations with those of the GNN, enabling the model to incorporate low-level circuit characteristics. VeriDistill outperforms state-of-the-art

methods on large-scale Verilog datasets, demonstrating superior accuracy and generalizability, particularly on out-of-distribution circuits.

Agentic-HLS [181] presents an HLS system that combines agentic reasoning and LLMs to enhance the automation and accuracy of HLS DSE and performance prediction. The system integrates CoT prompting, a GNN-based HARP encoder, and an agentic predictor-criticiser framework to evaluate and refine design predictions iteratively. Agentic-HLS leverages LLMs to analyze source code, reason about pragma impacts on control/data flow graphs, and predict key performance metrics such as latency, resource utilization (BRAM, LUT, FF, DSP), and design validity. By incorporating HARP graph embeddings, the system achieves significant improvements in classification and regression tasks for design optimization. Experimental results demonstrate that the fine-tuned HARP model achieves the best RMSE score (2.82), while the integrated Agentic-HLS system achieves a competitive RMSE of 4.21. The study highlights the limitations of CoT prompting in smaller models and emphasizes the potential for improved reasoning and performance prediction with larger models, such as GPT-4o.

7 Relevant Surveys

He et al. [182] introduce applications that apply LLM into circuit design and the EDA flow, mainly including three categories: on circuit design with LLM, design flow with LLM, and on debugging in Verilog. Several concerns regarding ethical issues and others, as well as possible future directions, are further analyzed.

Zhong et al. [183] survey the application of LLMs in the EDA field, compiled into three aspects: assistant chatbot, HDL and script generation, and HDL verification and analysis. Future research directions are highlighted, such as addressing long feedback of PPA by domain-specific PPA dataset, utilizing PPA as feedback, and proper prompt engineering techniques.

Pan et al. [184] provide a comprehensive review of the applications, methodologies, and challenges of integrating LLMs into Electronic EDA. They explore the use of LLMs across key stages such as system-level design, RTL design, logic synthesis, physical design, and analog circuit design, highlighting advancements in HDL generation, script automation, multi-modal feature representation, and debugging frameworks. The article categorizes techniques like supervised fine-tuning, DAPT, prompt engineering, and autonomous agent frameworks, emphasizing their impact on improving syntax correctness, functional accuracy, and design efficiency. It also discusses the academic infrastructure supporting LLM4EDA research, including datasets, benchmarks, and open-source models, while addressing bottlenecks such as scalability, confidentiality, and integration into legacy workflows. The survey concludes by identifying future opportunities in multi-modal training, explainable AI, and domain-specific LLM development to revolutionize EDA workflows and improve design outcomes.

Alsaqer et al. [185] provide a comprehensive survey on the potential of LLMs in revolutionizing hardware design, focusing on their applications in HDL code generation and hardware security. It highlights the transformative role of LLMs like GPT in automating labor-intensive processes, such as Verilog code generation, design verification, and security vulnerability detection, thereby enhancing productivity and reducing design costs. The study reviews 35 recent works, classifying them into two main categories: generating Verilog code for hardware design entry and identifying or fixing security vulnerabilities in hardware. Key insights include the effectiveness of fine-tuned LLMs for generating syntactically and functionally correct HDL code, the potential of LLMs to automate assertion generation for formal verification, and their ability to debug and repair security-relevant hardware bugs. Despite these advancements, challenges such as hallucinations, limited domain-specific datasets, and scalability issues persist. The article identifies future research opportunities, including the development of domain-specific and open-source LLMs,

expanding their role in analog circuit design, HW-SW co-design, and formal specification generation.

Xu et al. [186] explore the integration of LLMs into EDA workflows to enhance efficiency and reduce manual effort in hardware design. It reviews current advancements of LLMs in HDL generation, code debugging, verification, and physical implementation, highlighting their potential to automate complex tasks such as RTL writing, testbench generation, and design optimization. The article also identifies future challenges and opportunities, such as multi-modal analysis, automated debugging, and seamless integration of EDA tools, emphasizing the transformative potential of LLMs to create intelligent EDA agents capable of fully automating the hardware design process while addressing challenges like scalability, hallucinations, and data privacy.

He et al. [187] explore the application of LLMs in EDA, covering their roles as both assistants and autonomous agents. The survey calls for future research on further customizing and fine-tuning these models for specific EDA applications, improving their interpretability and explainability, and integrating them more seamlessly into existing design environments. It also argues that the development of open-source datasets and benchmarks will be crucial for advancing the state of the art and ensuring that the benefits of LLMs are accessible to a broader community of designers and researchers.

Wang et al. [188] examine the application of LLMs, **large multimodal models** (**LMMs**), and **large circuit models** (**LCMs**)—collectively referred to as LxMs—in chip design, with a focus on both the opportunities and security risks. They provide an overview of state-of-the-art LLM tasks, framework designs, and training schemes, while also addressing critical issues such as representation alignment, optimization, design automation, and the security of LxMs themselves. The work calls for further research into these areas and compiles resources on LLMs for chip design, including applications, security concerns, and open-source tools, which are publicly available.

Saha et al. [189] explore the integration of LLMs into SoC security, addressing the growing complexity and ubiquity of SoC designs. They highlight the limitations of current security solutions and proposes LLMs as a means to achieve more efficient, scalable, and adaptable security verification. The research covers various applications of LLMs in SoC security, including vulnerability insertion, security assessment, security verification, and countermeasure development. The article also identifies key challenges, such as the lack of domain-specific training data and the difficulty in handling large-scale designs, while offering potential solutions and discussing the future prospects of LLMs in this field.

Akyash et al. [190] study the application of LLMs for enhancing hardware security, particularly focusing on RTL designs. The methodology involves two primary approaches, namely prompt engineering and fine-tuning. The study also examines the use of LLMs in automating the detection and mitigation of security vulnerabilities, including the generation of formal verification testbenches and the refinement of prompts to improve the accuracy and completeness of generated SVAs. Additionally, the article discusses the integration of concepts like graphs and **Abstract Syntax Trees** (**ASTs**) into LLMs to better handle the structural nuances of RTL code.

Chen et al. [191] call for the creation of dedicated foundation models for circuits, namely, the LCM, where each design stage of the EDA flow is considered a separate modality and requires a specific representation learning strategy to embed and align the available circuit characteristics.

Qayyum et al. [192] explore the application of LLMs in the domain of hardware verification. It provides a comprehensive overview of state-of-the-art frameworks, tools, and techniques used to leverage LLMs for tasks such as generating hardware assertions, detecting bugs, and formal verification of hardware designs. The authors highlight the challenges posed by the inherent non-determinism and inconsistency of LLM-generated properties across different HDLs. The study also evaluates the use of LLMs integrated with frameworks like LangChain and RAG for verification

workflows, demonstrating their ability to generate, refine, and evaluate formal properties. Preliminary results show significant variability in the consistency of LLM-generated properties across HDLs, emphasizing the need for domain-specific fine-tuning and formal reasoning integration. The article concludes by identifying key future research directions, including improving property completeness, enhancing reasoning capabilities, and developing EDA-specific embeddings, to make LLMs more reliable and effective for hardware verification tasks.

Kahng [193] outlooks the 'next wave' for AI/ML in physical design, including the onrush of generative AI. He introduces that the near term will witness leveraging LLMs along three main vectors, namely design assistant to answer queries, prompt-based tool flows for live feedback, and graph generative models for incremental designs. He points out that these vectors can potentially revolutionize designer productivity, but perhaps not end quality of outcomes in physical design.

8 A 5-level Roadmap to LLM-empowered Autonomous EDA

This survey has outlined the transformative potential of LLMs in the domain of EDA. We have explored how these powerful language models are evolving from basic assistants to increasingly sophisticated partners in the design process. To guide future research and development, we propose a 5-level roadmap that charts a path toward a future of intelligent and autonomous EDA:

— L1 (Basic Application): This level represents the initial stage of LLM integration into EDA. LLMs act primarily as information retrieval and processing tools, assisting designers with tasks that do not directly involve the design process itself. They can search through vast libraries of documentation, answer questions about design principles and methodologies, summarize technical papers, and even provide natural language interfaces for existing EDA tools. This level is about making EDA knowledge and resources more accessible and user-friendly. Imagine asking an LLM to find the best documentation on a specific transistor model or to explain the concept of a certain design constraint in plain English. These are examples of the tasks that Level 1 LLMs excel at.

Challenges.

— Ensuring that the information retrieved or summarized by LLMs is accurate, relevant, and up-to-date is critical, since misinformation or outdated resources could mislead designers.
— LLMs must be fine-tuned to handle the technical jargon, concepts, and structure of EDA data, which is often highly specialized and domain-specific.
— Seamlessly integrating LLMs with existing EDA tools and workflows requires developing user-friendly interfaces and APIs.

Opportunities.

— LLMs can make complex EDA knowledge accessible to non-experts, fostering collaboration across interdisciplinary teams.
— By handling tedious information retrieval, LLMs free designers to focus on creative and strategic tasks.
— LLMs can be used as teaching aids, explaining complex design principles in plain language to students and new engineers.

— L2 (Helpers): Level 2 moves beyond basic information retrieval and enters the realm of task automation. LLMs begin to actively assist designers by generating scripts for repetitive tasks, automating common design workflows, and even suggesting code completions. For example, an LLM can generate a script to set up a simulation based on a specific design requirement or suggest a missing line of code to fix a DRC error. This level focuses on making designers more efficient by eliminating tedious and time-consuming tasks. Think of it as having a programming assistant that can handle the repetitive grunt work while you focus on the more creative aspects of design.

Challenges.
- Automating workflows requires LLMs to handle edge cases and errors gracefully to avoid introducing bugs or disrupting the design process.
- EDA workflows vary widely across tools, domains, and teams, creating a need for LLMs to adapt to different environments and requirements.
- Automated workflows could inadvertently expose sensitive design data if not properly secured.

Opportunities.
- Automating repetitive tasks can significantly reduce development time and improve workflow efficiency.
- LLMs can help enforce best practices in script generation and workflows, leading to more consistent results across teams.
- LLMs can learn from user feedback to provide increasingly tailored and effective automation solutions.

- L3 (Active Design Participation): Level 3 is where LLMs become true partners in the design process. They go beyond simple automation and begin to actively participate in design decisions, offering suggestions, identifying design patterns, and even recommending components based on specific constraints. For example, an LLM could suggest a layout optimization based on an analysis of signal integrity or recommend a specific capacitor based on power consumption requirements. Level 3 LLMs can understand design context and constraints, providing valuable insights and guiding the design process toward more optimal solutions.

  Challenges.
  - LLMs must interpret design requirements, constraints, and tradeoffs across multiple domains, such as analog, digital, and mixed-signal design.
  - Achieving this level requires LLMs to process and generate insights from diverse data formats, including schematics, layouts, and performance metrics.
  - The complexity of modern designs demands an in-depth understanding of system-level interactions that is challenging for LLMs.

  Opportunities.
  - LLMs can assist designers in exploring alternative solutions, identifying tradeoffs, and optimizing for performance metrics.
  - By acting as a design partner, LLMs can enhance creativity, helping designers explore novel ideas they might not have considered.
  - LLMs can provide immediate feedback during the design process, reducing iteration cycles and improving the quality of final designs.

- L4 (Guided Optimization and Decision Making): At Level 4, LLMs transition from collaborators to autonomous agents. They can now perform complex design tasks with minimal human intervention, exploring vast design spaces, making tradeoffs based on predefined constraints, and continuously adapting to evolving requirements. Imagine an LLM that can explore a vast range of circuit topologies, optimize for performance and power consumption, and present the best-performing design based on predefined criteria. This level represents a significant leap towards truly intelligent and automated EDA.

  Challenges.
  - Developing advanced optimization techniques that can balance tradeoffs across multiple conflicting objectives (e.g., power, performance, area) is a major hurdle.

— Handling large-scale design spaces and high-dimensional optimization problems requires significant computational resources and efficient algorithms.
— Ensuring the correctness of LLM-generated designs is critical, as errors at this stage could have catastrophic downstream consequences.

Opportunities.
— LLMs can explore a much larger range of design possibilities than humans, potentially uncovering innovative solutions.
— By taking over time-intensive optimization tasks, LLMs can dramatically accelerate the design cycle.
— LLMs with real-time feedback loops can learn from past results to continuously refine their optimization strategies.

— L5 (Autonomous Agent and Super-Human Performance): Level 5 represents the ultimate goal of LLM integration into EDA. LLMs at this level become autonomous design agents, potentially surpassing human capabilities. They can perform tasks such as generative design, synthesizing new circuits and systems, and even iterating and optimizing designs in a completely unsupervised manner. This level is about pushing the boundaries of design innovation, exploring uncharted design spaces, and potentially leading to breakthroughs that were previously unimaginable. Imagine an LLM that can invent new circuit architectures, optimize for performance beyond human intuition, and even learn and adapt to new design challenges without human guidance.

Challenges.
— Ensuring that autonomous LLMs consistently produce reliable and safe designs is paramount, especially when operating without human supervision.
— Autonomous agents raise ethical questions about accountability, intellectual property ownership, and the potential misuse of these systems.
— Achieving this level requires unprecedented computational power and access to massive, high-quality datasets.

Opportunities.
— Autonomous LLMs can explore uncharted design spaces, leading to revolutionary advancements in circuit architectures and EDA methodologies.
— Fully autonomous systems can operate continuously, dramatically reducing time-to-market for new designs.
— LLMs at this level could innovate by integrating knowledge and methodologies from other domains, such as biology or quantum computing, into EDA.

Figure 12 illustrates the proposed 5-level roadmap (including Level 0 meaning no LLM).

This 5-level roadmap provides a structured approach to the integration and advancement of LLMs in EDA. As we move from basic applications to fully autonomous agents, each level builds upon the previous one, increasing the complexity and depth of integration, representing a significant advancement in the role and capabilities of LLMs in EDA. By leveraging the power of these sophisticated language models, we can accelerate innovation, improve design efficiency, and unlock new frontiers in electronic design. The future of EDA lies in embracing the collaborative potential of LLMs, where human ingenuity and AI capabilities work together to push the boundaries of what's possible.

9 Discussions

Despite their transformative potential, applying LLMs to EDA faces several key challenges that must be addressed to ensure their effective and ethical deployment. These challenges include concerns

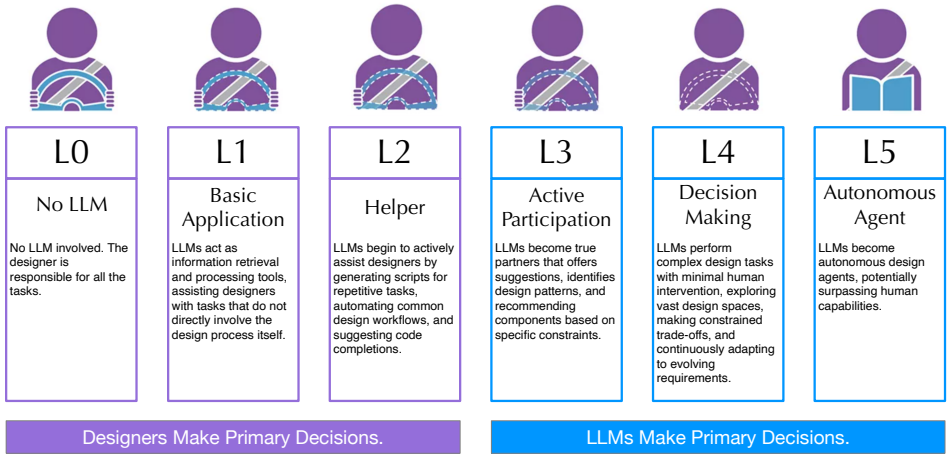


Fig. 12. The 5-level roadmap to LLM-empowered autonomous EDA. Design adapted from [194].

about data privacy, the reliability of the models, the impact on the workforce, and dataset quality and diversity. Below, we discuss these issues in detail and outline potential solutions.

Data Privacy. EDA workflows often involve proprietary information, including sensitive circuit designs, **intellectual property** (**IP**), and fabrication details. Leveraging LLMs typically requires large datasets, which may expose private or confidential data during model training or inference. Training LLMs on sensitive data without strong privacy guarantees risks disclosing proprietary IPs, violating **non-disclosure agreements** (**NDAs**), and exposing trade secrets. Besides, many commercial LLMs (e.g., GPT models) rely on cloud-based APIs, which may lead to concerns about data leakage during inference.

In particular, Extraction attacks [195] typically refer to attempts by adversaries to extract sensitive information or insights from ML models or their associated data. It has been shown that LLMs memorize and leak training data [196], even without a surrogate dataset [197]. Defense methods either improve training, such as corpora de-identification [198] and safety alignment [21], or are deployed during inference, such as malicious detection [199].

Reliability. Model reliability refers to the ability of an LLM to consistently produce accurate, trustworthy, and predictable outputs. Unreliable behavior, such as generating incorrect, incomplete, or nonsensical results, can have significant consequences, including design failures, wasted resources, or even security vulnerabilities. LLMs often exhibit issues such as hallucinations [200], where the model generates fabricated or inaccurate outputs, and inconsistencies [201], where it provides different answers to similar queries. Such behavior can lead to a lack of trust in the model's outputs, particularly in mission-critical applications. Additionally, the black-box nature of LLMs makes it difficult to interpret their decisions or trace the source of errors, further compounding reliability concerns.

Self-reflection [202] has been proposed to tackle the hallucination challenge through this feedback process. Retrieval Augmentation also reduces hallucination in knowledge-grounded conversations [203]. CoT [25] is a series of intermediate reasoning steps that significantly improves the ability of LLMs to perform complex reasoning. LLM uncertainty estimation [204] has also been greatly explored.

Workforce. The adoption of LLMs in EDA may disrupt traditional workflows, raising concerns about the potential impact on the workforce. The integration of LLMs into industry operations

offers numerous benefits [205], such as automation of repetitive tasks, improving data driven decision-making, driving cost reductions and improving operational efficiency, and enhancing the customer experience through highly personalized interactions. However, LLMs also face ethical and societal challenges, including potential job displacement and the risk of misinformation, as they can be used to generate misleading content at scale. A recent research [206] shows that over 46% of jobs in software development could have over half their tasks affected by LLMs with simple interfaces and general training. From a technical perspective, it is demonstrated that explainable AI improves human-AI collaboration, allowing domain experts to validate AI predictions against their own domain knowledge [207].

Dataset Quality. The quality and diversity of training datasets directly impact the performance and generalizability of LLMs in EDA tasks. However, sourcing comprehensive datasets for EDA remains a significant challenge. EDA datasets, especially for AMS circuits, are scarce due to proprietary restrictions and limited open-source contributions. LLMs trained on unbalanced datasets may exhibit biases, leading to suboptimal performance for niche applications or novel designs. As for LLM pretraining, it is predicted in [208] that models will be trained on datasets roughly equal in size to the available stock of public human text data between 2026 and 2032.

Generating high-quality synthetic data may pave the way for developing next-gen LLMs [209]. It may worth exploring using synthetic data generation techniques (e.g., procedural generation of SPICE netlists or Verilog code) to create diverse and scalable datasets. Data augmentation [210] is a pivotal technique for enhancing model performance by diversifying training examples without the need for additional data collection. Self-instruct [211] represents a bootstrapping method for inducing instruction following capabilities with minimal human-labeled data, which might be extended to dataset generation.

10 Conclusion

In this comprehensive survey, we have systematically reviewed the application of LLMs in the domain of EDA, categorizing the existing literature into four primary areas: code generation, verification and debugging, knowledge representation, and optimization/modeling. Our analysis has highlighted the significant potential of LLMs to revolutionize various aspects of the EDA process, from design synthesis and optimization to verification and documentation. Through code generation, LLMs automate the creation of HDLs, system-level designs, and flow scripts, significantly accelerating prototyping and reducing manual effort. In verification and debugging, they enhance design correctness by automating testbench generation, assertion synthesis, fault localization, and error correction, streamlining the process of detecting and resolving functional and security issues. Knowledge representation leverages the reasoning capabilities of LLMs to encode and retrieve design knowledge, aiding in specification understanding, bug classification, and efficient decision-making through QA systems. Finally, in optimization/modeling, LLMs enable improvements in design metrics such as PPA by automating circuit sizing, layout refinement, and workflow optimization. Together, these capabilities highlight the potential of LLMs to enhance productivity, improve design quality, and drive innovation in EDA, offering a new paradigm for tackling complex design and engineering challenges. However, realizing their full potential requires addressing key challenges, including data privacy, model reliability, dataset quality, and workforce adaptation, ensuring their deployment is both effective and ethical.

We have proposed a 5-level roadmap to guide the progressive integration and advancement of LLMs in EDA. By following the 5-level roadmap and addressing the associated challenges, we can unlock new levels of productivity, innovation, and efficiency in the EDA domain. As the technology continues to evolve, the potential for LLMs to transform the field of EDA is immense, and we look forward to the exciting developments and breakthroughs that lie ahead.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[2] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv:2302.13971. Retrieved from https://arxiv.org/abs/2302.13971

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. arXiv:2309.16609. Retrieved from https://arxiv.org/abs/2309.16609

[4] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv:2501.12948. Retrieved from https://arxiv.org/abs/2501.12948

[5] Ali Madani, Ben Krause, Eric R. Greene, Subu Subramanian, Benjamin P. Mohr, James M. Holton, Jose Luis Olmos, Caiming Xiong, Zachary Z. Sun, Richard Socher, et al. 2023. Large language models generate functional protein sequences across diverse families. *Nature Biotechnology* 41, 8 (2023), 1099–1106.

[6] Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. 2023. Bloomberggpt: A large language model for finance. arXiv:2303.17564. Retrieved from https://arxiv.org/abs/2303.17564

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[8] Jiaxi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. 2023. Chatlaw: Open-source legal large language model with integrated external knowledge bases. arXiv:2306.16092. Retrieved from https://arxiv.org/abs/2306.16092

[9] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, et al. 2019. Toward an open-source digital flow: First learnings from the openroad project. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–4.

[10] Xingquan Li, Zengrong Huang, Simin Tao, Zhipeng Huang, Chunan Zhuang, Hao Wang, Yifan Li, Yihang Qiu, Guojie Luo, Huawei Li, et al. 2024. iEDA: An open-source infrastructure of EDA. In *Proceedings of the 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC'24)*. IEEE, 77–82.

[11] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. 2021. Machine learning for electronic design automation: A survey. *ACM Transactions on Design Automation of Electronic Systems* 26, 5 (2021), 1–46.

[12] Bei Yu. 2023. Machine learning in EDA: When and how. In *Proceedings of the 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD'23)*. IEEE, 1–6.

[13] Siting Liu, Qi Sun, Peiyu Liao, Yibo Lin, and Bei Yu. 2021. Global placement with deep learning-enabled explicit routability optimization. In *Proceedings of the 2021 Design, Automation and Test in Europe Conference and Exhibition (DATE'21)*. IEEE, 1821–1824.

[14] Ziyi Wang, Chen Bai, Zhuolun He, Guangliang Zhang, Qiang Xu, Tsung-Yi Ho, Bei Yu, and Yu Huang. 2022. Functionality matters in netlist representation learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 61–66.

[15] Ziyi Wang, Siting Liu, Yuan Pu, Song Chen, Tsung-Yi Ho, and Bei Yu. 2023. Restructure-tolerant timing prediction via multimodal fusion. In *Proceedings of the 2023 60th ACM/IEEE Design Automation Conference (DAC'23)*. IEEE, 1–6.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017).

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. arXiv:2001.08361. Retrieved from https://arxiv.org/abs/2001.08361

[18] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. arXiv:2206.07682. Retrieved from https://arxiv.org/abs/2206.07682

[19] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://arxiv.org/abs/1810.04805

[20] Thomas Wang, Adam Roberts, Daniel Hesslow, Teven Le Scao, Hyung Won Chung, Iz Beltagy, Julien Launay, and Colin Raffel. 2022. What language model architecture and pretraining objective works best for zero-shot generalization?. In *Proceedings of the International Conference on Machine Learning*. PMLR, 22964–22984.

[21] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[22] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems* 36 (2023), 53728–53741.

[23] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. arXiv:2106.09685. Retrieved from https://arxiv.org/abs/2106.09685

[24] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* 36 (2023), 10088–10115.

[25] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[26] Tom B. Brown. 2020. Language models are few-shot learners. arXiv:2005.14165. Retrieved from https://arxiv.org/abs/2005.14165

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[28] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. arXiv:2301.00234. Retrieved from https://arxiv.org/abs/2301.00234

[29] Hao Yu, Aoran Gan, Kai Zhang, Shiwei Tong, Qi Liu, and Zhaofeng Liu. 2024. Evaluation of retrieval-augmented generation: A survey. arXiv:2405.07437. Retrieved from https://arxiv.org/abs/2405.07437

[30] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2023), 68539–68551.

[31] Hammond Pearce, Benjamin Tan, and Ramesh Karri. 2020. Dave: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. 27–32.

[32] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking large language models for automated verilog rtl code generation. In *Proceedings of the 2023 Design, Automation and Test in Europe Conference and Exhibition (DATE'23)*. IEEE, 1–6.

[33] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–31.

[34] Michael Tomlinson, Joe Li, and Andreas Andreou. 2024. Designing silicon brains using LLM: Leveraging chatgpt for automated description of a spiking neuron array. In *Proceedings of the 2024 Argentine Conference on Electronics (CAE'24)*. IEEE, 154–159.

[35] Paola Vitolo, George Psaltakis, Michael Tomlinson, Gian Domenico Licciardo, and Andreas G. Andreou. 2024. Natural language to verilog: Design of a recurrent spiking neural network using large language models and ChatGPT. arXiv:2405.01419. Retrieved from https://arxiv.org/abs/2405.01419

[36] Wenhao Sun, Bing Li, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, and Ulf Schlichtmann. 2024. Classification-based automatic HDL code generation using LLMs. arXiv:2407.18326. Retrieved from https://arxiv.org/abs/2407.18326

[37] Wenhao Sun, Bing Li, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, and Ulf Schlichtmann. 2025. Paradigm-based automatic HDL code generation using LLMs. arXiv:2501.12702. Retrieved from https://arxiv.org/abs/2501.12702

[38] Bardia Nadimi and Hao Zheng. 2024. A multi-expert large language model architecture for verilog code generation. arXiv:2404.08029. Retrieved from https://arxiv.org/abs/2404.08029

[39] Bhuvnesh Kumar, Saurav Nanda, Ganapathy Parthasarathy, Pawan Patil, Austin Tsai, and Parivesh Choudhary. 2024. HDL-GPT: High-quality HDL is all you need. arXiv:2407.18423. Retrieved from https://arxiv.org/abs/2407.18423

[40] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. BetterV: Controlled verilog generation with discriminative guidance. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[41] Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, et al. 2024. CodeV: Empowering LLMs for verilog generation through multi-level summarization. arXiv:2407.10424. Retrieved from https://arxiv.org/abs/2407.10424

[42] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2024. VerilogCoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (AST)-based waveform tracing tool. arXiv:2408.08927. Retrieved from https://arxiv.org/abs/2408.08927

[43] Yiyao Yang, Fu Teng, Pengju Liu, Mengnan Qi, Chenyang Lv, Ji Li, Xuhong Zhang, and Zhezhi He. 2025. HaVen: Hallucination-mitigated LLM for verilog code generation aligned with HDL engineers. arXiv:2501.04908. Retrieved from https://arxiv.org/abs/2501.04908

[44] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. 2024. Transferable pre-synthesis ppa estimation for rtl designs with data augmentation techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44 (2024), 200–213.

[45] Prashanth Vijayaraghavan, Apoorva Nitsure, Charles Mackin, Luyao Shi, Stefano Ambrogio, Arvind Haran, Viresh Paruthi, Ali Elzein, Dan Coops, David Beymer, et al. 2024. Chain-of-descriptions: Improving code LLMs for VHDL code generation and summarization. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–10.

[46] Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. 2024. AutoVCoder: A systematic framework for automated verilog code generation using LLMs. arXiv:2407.18333. Retrieved from https://arxiv.org/abs/2407.18333

[47] Madhav Nair, Rajat Sadhukhan, and Debdeep Mukhopadhyay. 2023. Generating secure hardware using chatgpt resistant to cwes. *Cryptology ePrint Archive* (2023).

[48] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2023. Autochip: Automating hdl generation using llm feedback. arXiv:2311.04887. Retrieved from https://arxiv.org/abs/2311.04887

[49] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. arXiv:2305.14019. Retrieved from https://arxiv.org/abs/2305.14019

[50] Kaiyan Chang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, Xiaowei Li, and Ying Wang. 2023. Improving large language model hardware generating quality through post-llm search. *Proceedings of the Machine Learning for Systems* (2023).

[51] Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahua Lin, Xingcheng Zhang, et al. 2024. OriGen: Enhancing RTL code generation with code-to-code augmentation and self-reflection. arXiv:2407.16237. Retrieved from https://arxiv.org/abs/2407.16237

[52] Mubashir ul Islam, Humza Sami, Pierre-Emmanuel Gaillardon, and Valerio Tenace. 2024. AIvril: AI-Driven RTL generation with verification in-the-loop. arXiv:2409.11411. Retrieved from https://arxiv.org/abs/2409.11411

[53] Hanxian Huang, Zhenghan Lin, Zixuan Wang, Xin Chen, Ke Ding, and Jishen Zhao. 2024. Towards LLM-powered verilog RTL assistant: Self-verification and self-correction. arXiv:2406.00115. Retrieved from https://arxiv.org/abs/2406.00115

[54] Kiran Thorat, Jiahui Zhao, Yaotian Liu, Hongwu Peng, Xi Xie, Bin Lei, Jeff Zhang, and Caiwen Ding. 2023. Advanced language model-driven verilog development: Enhancing power, performance, and area optimization in code synthesis. arXiv:2312.01022. Retrieved from https://arxiv.org/abs/2312.01022

[55] Bowei Wang, Qi Xiong, Zeqing Xiang, Lei Wang, and Renzhi Chen. 2025. RTLSquad: Multi-agent based interpretable RTL design. arXiv:2501.05470. Retrieved from https://arxiv.org/abs/2501.05470

[56] Andre Nakkab, Sai Qian Zhang, Ramesh Karri, and Siddharth Garg. 2024. Rome was not built in a single step: Hierarchical prompting for LLM-based chip design. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–11.

[57] Xi Wang, Gwok-Waa Wan, Sam-Zaak Wong, Layton Zhang, Tianyang Liu, Qi Tian, and Jianmin Ye. 2024. ChatCPU: An agile CPU design and verification platform with LLM. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[58] Jinwei Tang, Jiayin Qin, Kiran Thorat, Chen Zhu-Tian, Yu Cao, and Caiwen Ding. 2024. HiVeGen–Hierarchical LLM-based verilog generation for scalable chip design. arXiv:2412.05393. Retrieved from https://arxiv.org/abs/2412.05393

[59] Ning Wang, Bingkun Yao, Jie Zhou, Xi Wang, Zhe Jiang, and Nan Guan. 2024. Large language model for verilog generation with golden code feedback. arXiv:2407.18271. Retrieved from https://arxiv.org/abs/2407.18271

[60] Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. 2024. Make every move count: Llm-based high-quality rtl code generation using mcts. arXiv:2402.03289. Retrieved from https://arxiv.org/abs/2402.03289

[61] Kaiyan Chang, Zhirong Chen, Yunhao Zhou, Wenlong Zhu, kun wang, Haobo Xu, Cangyuan Li, Mengdi Wang, Shengwen Liang, Huawei Li, Yinhe Han, and Ying Wang. 2024. Natural language is not enough: Benchmarking multi-modal generative AI for Verilog generation. In *Proceedings of the ICCAD'24*. DOI: https://doi.org/10.1145/3676536.3676679

[62] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD'23)*. IEEE, 1–8.

[63] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. Rtllm: An open-source benchmark for design rtl generation with large language model. In *Proceedings of the 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC'24)*. IEEE, 722–727.

[64] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2023. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. arXiv:2312.08617. Retrieved from https://arxiv.org/abs/2312.08617

[65] Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. 2024. CraftRTL: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. arXiv:2409.12993. Retrieved from https://arxiv.org/abs/2409.12993

[66] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2024. Revisiting VerilogEval: Newer LLMs, in-context learning, and specification-to-RTL tasks. arXiv:2408.11053. Retrieved from https://arxiv.org/abs/2408.11053

[67] Prashanth Vijayaraghavan, Luyao Shi, Stefano Ambrogio, Charles Mackin, Apoorva Nitsure, David Beymer, and Ehsan Degan. 2024. VHDL-Eval: A framework for evaluating large language models in VHDL code generation. arXiv:2406.04379. Retrieved from https://arxiv.org/abs/2406.04379

[68] Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024. OpenLLM-RTL: Open dataset and benchmark for LLM-Aided design RTL generation. In *Proceedings of the 2024 IEEE/ACM International Conference on Computer Aided Design (ICCAD'24)*.

[69] Weimin Fu, Shijie Li, Yifang Zhao, Kaichen Yang, Xuan Zhang, Yier Jin, and Xiaolong Guo. 2024. A generalize hardware debugging approach for large language models semi-syntectic datasets. *Authorea Preprints* (2024).

[70] Vladyslav Romashchenko, Michael Brutscheck, and Ingo Chmielewski. 2024. Investigation and implementation of AI-HDLCoder for automated VHDL code synthesis and code generation for hardware SoC development. In *Proceedings of the 2024 35th Irish Signals and Systems Conference (ISSC'24)*. IEEE, 1–6.

[71] Emil Goh, Maoyang Xiang, I. Wey, and T. Hui Teo. 2024. From English to ASIC: Hardware implementation with large language model. arXiv:2403.07039. Retrieved from https://arxiv.org/abs/2403.07039

[72] Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan (Celine) Lin. 2024. MG-Verilog: Multi-grained dataset towards enhanced LLM-assisted verilog generation. In *Proceedings of the 1st IEEE International Workshop on LLM-Aided Design (LAD'24)*.

[73] Christopher Batten, Nathaniel Pinckney, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2024. PyHDL-Eval: An LLM evaluation framework for hardware design using python-embedded DSLs. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–17.

[74] Ahmed Allam and Mohamed Shalan. 2024. RTL-Repo: A benchmark for evaluating LLMs on large-scale RTL design projects. arXiv:2405.17378. Retrieved from https://arxiv.org/abs/2405.17378

[75] Deepak Vungarala, Mahmoud Nazzal, Mehrdad Morsali, Chao Zhang, Arnob Ghosh, Abdallah Khreishah, and Shaahin Angizi. 2024. SA-DS: A dataset for large language model-driven AI accelerator design generation. arXiv:2404.10875. Retrieved from https://arxiv.org/abs/2404.10875

[76] Manar Abdelatty, Jingxiao Ma, and Sherief Reda. 2025. MetRex: A benchmark for verilog code metric reasoning using LLMs. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC'25)*.

[77] Matthew DeLorenzo, Vasudev Gohil, and Jeyavijayan Rajendran. 2024. CreativEval: Evaluating creativity of LLM-Based hardware code generation. arXiv:2404.08806. Retrieved from https://arxiv.org/abs/2404.08806

[78] Sichao Yang and Ye Yang. 2024. FormalEval: A method for automatic evaluation of code generation via large language models. In *Proceedings of the 2024 2nd International Symposium of Electronics Design Automation (ISEDA'24)*. IEEE, 660–665.

[79] Mohammad Akyash and Hadi M. Kamali. 2025. SimEval: Investigating the similarity obstacle in LLM-based hardware code generation. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASPDAC'25)*. ACM, Tokyo, Japan, 1002–1007. DOI: http://doi.org/10.1145/3658617.3697624

[80] Kun Wang, Kaiyan Chang, Mengdi Wang, Xinqi Zou, Haobo Xu, Yinhe Han, and Ying Wang. 2025. RTLMarker: Protecting LLM-Generated RTL copyright via a hardware watermarking framework. arXiv:2501.02446. Retrieved from https://arxiv.org/abs/2501.02446

[81] Zheyu Yan, Yifan Qin, Xiaobo Sharon Hu, and Yiyu Shi. 2023. On the viability of using LLMs for SW/HW co-design: An example in designing CiM DNN accelerators. In *Proceedings of the 2023 IEEE 36th International System-on-Chip Conference (SOCC'23)*. IEEE, 1–6.

[82] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. 2023. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In *Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD'23)*. IEEE, 1–9.

[83] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-chat: Challenges and opportunities in conversational hardware design. In *Proceedings of the 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD'23)*. IEEE, 1–6.

[84] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2024. Evaluating LLMs for hardware design and test. arXiv:2405.02326. Retrieved from https://arxiv.org/abs/2405.02326

[85] Kimia Tasnia and Sazadur Rahman. 2025. OPL4GPT: An application space exploration of optimal programming language for hardware design by LLM. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC'25)*.

[86] Aditya Patra, Saroj Rout, and Arun Ravindran. 2024. AiEDA: Agentic AI design framework for digital ASIC system design. arXiv:2412.09745. Retrieved from https://arxiv.org/abs/2412.09745

[87] William Salcedo, Courtney McBeth, and Sara Achour. 2024. Leveraging generative AI for rapid design and verification of a vector processor SoC. *IEEE Design and Test* 41, 1 (2024), 8–18.

[88] Zhuolun He, Haoyuan Wu, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. 2023. ChatEDA: A large language model powered autonomous agent for EDA. In *Proceedings of the 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD'23)*. IEEE, 1–6.

[89] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. 2024. Chateda: A large language model powered autonomous agent for eda. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 10 (2024), 3184–3197.

[90] Haoyuan Wu, Haisheng Zheng, Zhuolun He, and Bei Yu. 2025. Divergent thoughts toward one goal: LLM-based multi-agent collaboration system for electronic design automation. arXiv:2502.10857. Retrieved from https://arxiv.org/abs/2502.10857

[91] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. Chipnemo: Domain-adapted llms for chip design. arXiv:2311.00176. Retrieved from https://arxiv.org/abs/2311.00176

[92] Utsav Sharma, Bing-Yue Wu, Sai Rahul Dhanvi Kankipati, Vidya A. Chhabria, and Austin Rovinski. 2024. OpenROAD-Assistant: An open-source large language model for physical design tasks. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–7.

[93] Luca Collini, Siddharth Garg, and Ramesh Karri. 2024. C2HLSC: Can LLMs bridge the software-to-hardware design gap? arXiv:2406.09233. Retrieved from https://arxiv.org/abs/2406.09233

[94] Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, and Bing Li. 2024. Automated C/C++ program repair for high-level synthesis via large language models. arXiv:2407.03889. Retrieved from https://arxiv.org/abs/2407.03889

[95] Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. 2024. HLSPilot: LLM-based high-level synthesis. arXiv:2408.06810. Retrieved from https://arxiv.org/abs/2408.06810

[96] Sneha Swaroopa, Rijoy Mukherjee, Anushka Debnath, and Rajat Subhra Chakraborty. 2024. Evaluating large language models for automatic register transfer logic generation via high-level synthesis. arXiv:2408.02793. Retrieved from https://arxiv.org/abs/2408.02793

[97] James T. Meech. 2023. Leveraging high-level synthesis and large language models to generate, simulate, and deploy a uniform random number generator hardware design. arXiv:2311.03489. Retrieved from https://arxiv.org/abs/2311.03489

[98] Yuchao Liao, Tosiron Adegbija, and Roman Lysecky. 2024. Are LLMs any good for high-level synthesis? arXiv:2408.10428. Retrieved from https://arxiv.org/abs/2408.10428

[99] Seyed Arash Sheikholeslam and Andre Ivanov. 2024. SynthAI: A multi agent generative AI framework for automated modular HLS design generation. arXiv:2405.16072. Retrieved from https://arxiv.org/abs/2405.16072

[100] Tianyang Liu, Qi Tian, Jianmin Ye, LikTung Fu, Shengchu Su, Junyan Li, Gwok-Waa Wan, Layton Zhang, Sam-Zaak Wong, Xi Wang, et al. 2024. ChatChisel: Enabling agile hardware design with large language models. In *Proceedings of the 2024 2nd International Symposium of Electronics Design Automation (ISEDA'24)*. IEEE, 710–716.

[101] Jiahao Gai, (Mark) Hao Chen, Zhican Wang, Hongyu Zhou, Wanru Zhao, Nicholas Lane, and Hongxiang Fan. 2025. Exploring code language models for automated HLS-based hardware generation: Benchmark, infrastructure and analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC'25)*.

[102] Yao Lai, Sungyoung Lee, Guojin Chen, Souradip Poddar, Mengkang Hu, David Z. Pan, and Ping Luo. 2024. AnalogCoder: Analog circuit design via training-free code generation. arXiv:2405.14918. Retrieved from https://arxiv.org/abs/2405.14918

[103] Jinyi Shen, Zihao Chen, Ji Zhuang, Jiangli Huang, Fan Yang, Li Shang, Zhaori Bi, Changhao Yan, Dian Zhou, and Xuan Zeng. 2024. Atelier: An automated analog circuit design framework via multiple large language model-based agents. *Authorea Preprints* (2024).

[104] Deepak Vungarala, Sakila Alam, Arnob Ghosh, and Shaahin Angizi. 2024. SPICEPilot: Navigating SPICE code generation and simulation with AI guidance. arXiv:2410.20553. Retrieved from https://arxiv.org/abs/2410.20553

[105] Yuchao Wu, Xiaofei Yu, Hao Chen, Yang Luo, Yeyu Tong, and Yuzhe Ma. 2025. PICBench: Benchmarking LLMs for photonic integrated circuits design. arXiv:2502.03159. Retrieved from https://arxiv.org/abs/2502.03159

[106] Zihao Chen, Jiangli Huang, Yiting Liu, Fan Yang, Li Shang, Dian Zhou, and Xuan Zeng. 2024. Artisan: Automated operational amplifier design via domain-specific large language model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[107] Chen-Chia Chang, Yikang Shen, Shaoze Fan, Jing Li, Shun Zhang, Ningyuan Cao, Yiran Chen, and Xin Zhang. 2024. LaMAGIC: Language-model-based topology generation for analog integrated circuits. In *Proceedings of the International Conference on Machine Learning (ICML'24)*. JMLR.org, Article 241, 10 pages.

[108] Jitendra Bhandari, Vineet Bhat, Yuheng He, Siddharth Garg, Hamed Rahmani, and Ramesh Karri. 2024. Auto-SPICE: Leveraging LLMs for dataset creation via automated SPICE netlist extraction from analog circuit diagrams. arXiv:2411.14299. Retrieved from https://arxiv.org/abs/2411.14299

[109] Ryoga Matsuo, Stefan Uhlich, Arun Venkitaraman, Andrea Bonetti, Chia-Yu Hsieh, Ali Momeni, Lukas Mauch, Augusto Capone, Eisaku Ohbuchi, and Lorenzo Servadei. 2024. Schemato–An LLM for netlist-to-schematic conversion. arXiv:2411.13899. Retrieved from https://arxiv.org/abs/2411.13899

[110] Bingyang Liu, Haoyi Zhang, Xiaohan Gao, Zichen Kong, Xiyuan Tang, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. LayoutCopilot: An LLM-powered multi-agent collaborative framework for interactive analog layout design. arXiv:2406.18873. Retrieved from https://arxiv.org/abs/2406.18873

[111] Ali Hammoud, Chetanya Goyal, Sakib Pathen, Arlene Dai, Anhang Li, Gregory Kielian, and Mehdi Saligane. 2024. Human language to analog layout using GLayout layout automation framework. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–7.

[112] Zixiao Wang, Yunheng Shen, Xufeng Yao, Wenqian Zhao, Yang Bai, Farzan Farnia, and Bei Yu. 2024. ChatPattern: Layout pattern customization via natural language. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*.

[113] Guojin Chen, Haoyu Yang, Bei Yu, and Haoxing Ren. 2024. Intelligent OPC engineer assistant for semiconductor manufacturing. arXiv:2408.12775. Retrieved from https://arxiv.org/abs/2408.12775

[114] Luis Francisco and Srini Arikati. 2024. LLM based physical verification runset generator. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–7.

[115] Chen-Chia Chang, Chia-Tung Ho, Yaguang Li, Yiran Chen, and Haoxing Ren. 2024. DRC-Coder: Automated DRC checker code generation using LLM autonomous agent. arXiv:2412.05311. Retrieved from https://arxiv.org/abs/2412.05311

[116] Chuyue Sun, Christopher Hahn, and Caroline Trippel. 2023. Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions. In *Proceedings of the 1st International Workshop on Deep Learning-aided Verification*.

[117] Muhammad Hassan, Sallar Ahmadi-Pour, Khushboo Qayyum, Chandan Kumar Jha, and Rolf Drechsler. 2024. Llm-guided formal verification coupled with mutation testing. In *Proceedings of the 2024 Design, Automation and Test in Europe Conference and Exhibition (DATE'24)*. IEEE, 1–2.

[118] FNU Aditi and Michael S. Hsiao. 2023. Validatable generation of system verilog assertions from natural language specifications. In *Proceedings of the 2023 5th International Conference on Transdisciplinary AI (TransAI'23)*. IEEE, 102–109.

[119] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. 2024. Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms. arXiv:2402.00386. Retrieved from https://arxiv.org/abs/2402.00386

[120] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. 2024. AutoBench: Automatic testbench generation and evaluation using LLMs for HDL design. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–10.

[121] Bhabesh Mali, Karthik Maddala, Vatsal Gupta, Sweeya Reddy, Chandan Karfa, and Ramesh Karri. 2024. Chiraag: Chatgpt informed rapid and automated assertion generation. In *Proceedings of the 2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI'24)*. IEEE, 680–683.

[122] Karthik Maddala, Bhabesh Mali, and Chandan Karfa. 2024. LAAG-RV: LLM assisted assertion generation for RTL design verification. In *Proceedings of the 2024 IEEE 8th International Test Conference India (ITC India'24)*. IEEE, 1–6.

[123] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. 2023. Using llms to facilitate formal verification of rtl. arXiv:2309.09437. Retrieved from https://arxiv.org/abs/2309.09437

[124] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. 2024. Correctbench: Automatic testbench generation with functional self-correction using llms for hdl design. arXiv:2411.08510. Retrieved from https://arxiv.org/abs/2411.08510

[125] Zixi Zhang, Greg Chadwick, Hugo McNally, Yiren Zhao, and Robert Mullins. 2023. Llm4dv: Using large language models for hardware test stimuli generation. arXiv:2310.04535. Retrieved from https://arxiv.org/abs/2310.04535

[126] Jitendra Bhandari, Johann Knechtel, Ramesh Narayanaswamy, Siddharth Garg, and Ramesh Karri. 2024. LLM-aided testbench generation and bug detection for finite-state machines. arXiv:2406.17132. Retrieved from https://arxiv.org/abs/2406.17132

[127] Yifei Deng, Renzhi Chen, Chao Xiao, Zhijie Yang, Yuanfeng Luo, Jingyue Zhao, Na Li, Zhong Wan, Yongbao Ai, Huadong Dai, et al. 2024. LLM-TG: Towards automated test case generation for processors using large language

models. In *Proceedings of the 2024 IEEE 42nd International Conference on Computer Design (ICCD'24)*. IEEE, 389–396.

[128] Ruiyang Ma, Yuxin Yang, Ziqian Liu, Jiaxi Zhang, Min Li, Junhua Huang, and Guojie Luo. 2024. VerilogReader: LLM-Aided hardware test generation. arXiv:2406.04373. Retrieved from https://arxiv.org/abs/2406.04373

[129] Sudipta Paria, Aritra Dasgupta, and Swarup Bhunia. 2023. Divas: An llm-based end-to-end framework for soc security analysis and policy-based protection. arXiv:2308.06932. Retrieved from https://arxiv.org/abs/2308.06932

[130] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. 2024. (Security) assertions by large language models. *IEEE Transactions on Information Forensics and Security* 19 (2024), 4374–4389. DOI: http://doi.org/10.1109/TIFS.2024.3372809

[131] Jayeeta Chaudhuri, Dhruv Thapar, Arjun Chaudhuri, Farshad Firouzi, and Krishnendu Chakrabarty. 2024. SPICED: Syntactical bug and trojan pattern identification in A/MS circuits using LLM-Enhanced detection. arXiv:2408.16018. Retrieved from https://arxiv.org/abs/2408.16018

[132] Avinash Ayalasomayajula, Rui Guo, Jingbo Zhou, Sujan Kumar Saha, and Farimah Farahmandi. 2024. LASP: LLM assisted security property generation for SoC verification. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–7.

[133] Samit Shahnawaz Miftah, Amisha Srivastava, Hyunmin Kim, and Kanad Basu. 2024. Assert-O: Context-based assertion optimization using LLMs. In *Proceedings of the Great Lakes Symposium on VLSI 2024*. 233–239.

[134] Shams Tarek, Dipayan Saha, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi. 2024. SoCureLLM: An LLM-driven approach for large-scale system-on-chip security verification and policy generation. *Cryptology ePrint Archive* (2024).

[135] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2023. Rtlfixer: Automatically fixing rtl syntax errors with large language models. arXiv:2311.16543. Retrieved from https://arxiv.org/abs/2311.16543

[136] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. arXiv:2210.03629. Retrieved from https://arxiv.org/abs/2210.03629

[137] Ke Xu, Jialin Sun, Yuchen Hu, Xinwei Fang, Weiwei Shan, Xi Wang, and Zhe Jiang. 2024. MEIC: Re-thinking RTL debug automation using LLMs. arXiv:2405.06840. Retrieved from https://arxiv.org/abs/2405.06840

[138] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing hardware security bugs with large language models. arXiv:2302.01215. Retrieved from https://arxiv.org/abs/2302.01215

[139] Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. 2023. LLM4SecHW: Leveraging domain-specific large language model for hardware debugging. In *Proceedings of the 2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST'23)*. IEEE, 1–6.

[140] Mohammad Akyash and Hadi Mardani Kamali. 2024. Self-HWDebug: Automation of LLM self-instructing for hardware security verification. arXiv:2405.12347. Retrieved from https://arxiv.org/abs/2405.12347

[141] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. Hdldebugger: Streamlining hdl debugging with large language models. arXiv:1701.00133. Retrieved from https://arxiv.org/abs/1701.00133

[142] Khushboo Qayyum, Muhammad Hassan, Sallar Ahmadi-Pour, Chandan Kumar Jha, and Rolf Drechsler. 2024. From bugs to fixes: HDL bug identification and patching using LLMs and RAG. In *Proceedings of the 2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 1–5.

[143] Bingkun Yao, Ning Wang, Jie Zhou, Xi Wang, Hong Gao, Zhe Jiang, and Nan Guan. 2024. Location is key: Leveraging large language model for functional bug localization in verilog. arXiv:2409.15186. Retrieved from https://arxiv.org/abs/2409.15186

[144] Lily Jiaxin Wan, Yingbing Huang, Yuhong Li, Hanchen Ye, Jinghua Wang, Xiaofan Zhang, and Deming Chen. 2024. Software/hardware co-design for llm and its application for design verification. In *Proceedings of the 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC'24)*. IEEE, 435–441.

[145] Jayeeta Chaudhuri, Dhruv Thapar, Arjun Chaudhuri, Farshad Firouzi, and Krishnendu Chakrabarty. 2025. SPICED+: Syntactical bug pattern identification and correction of Trojans in A/MS circuits using LLM-Enhanced detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 33, 4 (2025), 1118–1131.

[146] Weimin Fu, Shijie Li, Yifang Zhao, Haocheng Ma, Raj Dutta, Xuan Zhang, Kaichen Yang, Yier Jin, and Xiaolong Guo. 2024. Hardware phi-1.5 b: A large language model encodes hardware domain specific knowledge. In *Proceedings of the 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC'24)*. IEEE, 349–354.

[147] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023. Textbooks are all you need ii: Phi-1.5 technical report. arXiv:2309.05463. Retrieved from https://arxiv.org/abs/2309.05463

[148] Mengming Li, Wenji Fang, Qijun Zhang, and Zhiyao Xie. 2024. Specllm: Exploring generation and review of vlsi design specification with large language model. arXiv:2401.13266. Retrieved from https://arxiv.org/abs/2401.13266

[149] Boyu Han, Xinyu Wang, Yifan Wang, Junyu Yan, and Yidong Tian. 2023. New interaction paradigm for complex eda software leveraging gpt. arXiv:2307.14740. Retrieved from https://arxiv.org/abs/2307.14740

[150] Yuan Pu, Zhuolun He, Tairu Qiu, Haoyuan Wu, and Bei Yu. 2024. Customized retrieval augmented generation and benchmarking for EDA tool documentation QA. arXiv:2407.15353. Retrieved from https://arxiv.org/abs/2407.15353

[151] Yuan Pu, Zhuolun He, Yuqi Jiang, Tairu Qiu, Haoyuan Wu, Qi Sun, Cheng Zhuo, and Bei Yu. 2025. Customized retrieval augmented generation and benchmarking for EDA tool documentation QA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 1–1. DOI: http://doi.org/10.1109/TCAD.2025.3568776

[152] Aviral Kaintura, Shui Song Luar, and Indira Iyer Almeida. 2024. ORAssistant: A custom RAG-based conversational assistant for OpenROAD. arXiv:2410.03845. Retrieved from https://arxiv.org/abs/2410.03845

[153] Zhe Xiao, Xu He, Haoying Wu, Bei Yu, and Yang Guo. 2025. EDA-Copilot: A RAG-powered intelligent assistant for EDA tools. *ACM Transactions on Design Automation of Electronic Systems* (2025).

[154] Bing-Yue Wu, Utsav Sharma, Sai Rahul Dhanvi Kankipati, Ajay Yadav, Bintu Kappil George, Sai Ritish Guntupalli, Austin Rovinski, and Vidya A. Chhabria. 2024. EDA corpus: A large language model dataset for enhanced interaction with OpenROAD. arXiv:2405.06676. Retrieved from https://arxiv.org/abs/2405.06676

[155] Manar Abdelatty and Sherief Reda. 2024. HDLCopilot: Hardware design library querying with natural language. arXiv:2407.12749. Retrieved from https://arxiv.org/abs/2407.12749

[156] Supparesk Rittikulsittichai and Thitirat Siriborvornratanakul. 2024. An intelligent chatbot assistant for comprehensive troubleshooting guidelines and knowledge repository in printed circuit board production. *IEEE Transactions on Artificial Intelligence* 6, 5 (2024), 1259–1268.

[157] Luyao Shi, Michael Kazda, Bradley Sears, Nick Shropshire, and Ruchir Puri. 2024. Ask-EDA: A design assistant empowered by LLM, hybrid RAG and abbreviation de-hallucination. arXiv:2406.06575. Retrieved from https://arxiv.org/abs/2406.06575

[158] Ning Xu, Zhaoyang Zhang, Lei Qi, Wensuo Wang, Chao Zhang, Zihao Ren, Huaiyuan Zhang, Xin Cheng, Yanqi Zhang, Zhichao Liu, et al. 2024. ChipExpert: The open-source integrated-circuit-design-specific large language model. arXiv:2408.00804. Retrieved from https://arxiv.org/abs/2408.00804

[159] Chenhui Deng, Yunsheng Bai, and Haoxing Ren. 2024. ChipAlign: Instruction alignment in large language models for chip design via geodesic interpolation. arXiv:2412.19819. Retrieved from https://arxiv.org/abs/2412.19819

[160] Saruni Fernando, Robert Kunzelmann, Daniela Sánchez Lopera, Jad Al Halabi, and Wolfgang Ecker. 2024. Leveraging large language models for the automated documentation of hardware designs. In *Proceedings of the 2024 13th Mediterranean Conference on Embedded Computing (MECO'24)*. IEEE, 1–6.

[161] Li-C Wang. 2024. LLM-Assisted analytics in semiconductor test. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–7.

[162] Lejla Skelic, Yan Xu, Matthew Cox, Wenjie Lu, Tao Yu, and Ruonan Han. 2025. CIRCUIT: A benchmark for circuit interpretation and reasoning capabilities of LLMs. arXiv:2502.07980. Retrieved from https://arxiv.org/abs/2502.07980

[163] Yichen Shi, Zhuofu Tao, Yuhao Gao, Tianjia Zhou, Cheng Chang, Yaxing Wang, Bingyu Chen, Genhao Zhang, Alvin Liu, Zhiping Yu, et al. 2024. AMSnet-KG: A netlist dataset for LLM-based AMS circuit auto-design using knowledge graph RAG. arXiv:2411.13560. Retrieved from https://arxiv.org/abs/2411.13560

[164] Hong Cai Chen, Longchang Wu, Ming Gao, Lingrui Shen, Jiarui Zhong, and Yipin Xu. 2024. DocEDA: Automated extraction and design of analog circuits from documents with large language model. arXiv:2412.05301. Retrieved from https://arxiv.org/abs/2412.05301

[165] Siyu Qiu, Benjamin Tan, and Hammond Pearce. 2024. Explaining EDA synthesis errors with LLMs. arXiv:2404.07235. Retrieved from https://arxiv.org/abs/2404.07235

[166] Chengjie Liu, Yijiang Liu, Yuan Du, and Li Du. 2024. LADAC: Large language model-driven auto-designer for analog circuits. *Authorea Preprints* (2024).

[167] Chengjie Liu, Weiyu Chen, Anlan Peng, Yuan Du, Li Du, and Jun Yang. 2024. AmpAgent: An LLM-based multi-agent system for multi-stage amplifier schematic design from literature for process and performance porting. arXiv:2409.14739. Retrieved from https://arxiv.org/abs/2409.14739

[168] Guojin Chen, Keren Zhu, Seunggeun Kim, Hanqing Zhu, Yao Lai, Bei Yu, and David Z. Pan. 2024. LLM-Enhanced bayesian optimization for efficient analog layout constraint generation. arXiv:2406.05250. Retrieved from https://arxiv.org/abs/2406.05250

[169] Yuxuan Yin, Yu Wang, Boxun Xu, and Peng Li. 2024. ADO-LLM: Analog design bayesian optimization with in-context learning of large language models. arXiv:2406.18770. Retrieved from https://arxiv.org/abs/2406.18770

[170] Dimple Vijay Kochar, Hanrui Wang, Anantha Chandrakasan, and Xin Zhang. 2024. LEDRO: LLM-enhanced design space reduction and optimization for analog circuits. arXiv:2411.12930. Retrieved from https://arxiv.org/abs/2411.12930

[171] Haocheng Xu, Haotian Hu, and Sitao Huang. 2024. Optimizing high-level synthesis designs with retrieval-augmented large language models. In *Proceedings of the 2024 IEEE LLM Aided Design Workshop (LAD'24)*. IEEE, 1–5.

[172] Lin Chen, Ran Chen, Shoubo Hu, Xufeng Yao, Zhentao Tang, Shixiong Kai, Siyuan Xu, Mingxuan Yuan, Jianye Hao, Bei Yu, et al. 2025. PCBAgent: An agent-based framework for high-density printed circuit board placement. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC'25)*.

[173] Hao Wang, Jun Tu, Shenglong Bai, Jie Zheng, Weikang Qian, and Jienan Chen. 2024. Routing generative pre-trained transformers for printed circuit board. In *Proceedings of the 2024 2nd International Symposium of Electronics Design Automation (ISEDA'24)*. IEEE, 160–165.

[174] Jienan Chen, Jun Tu, Hao Wang, Jie Zheng, Hao Liu, Shenglong Bai, Xiantuo He, and Weikang Qian. 2024. GPCB routing: Generative pre-trained transformers based printed circuit board routing method. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44, 4 (2024), 1420–1433.

[175] Weihua Xiao, Venkata Sai Charan Putrevu, Raghu Vamshi Hemadri, Siddharth Garg, and Ramesh Karri. 2024. PrefixLLM: LLM-aided prefix circuit design. arXiv:2412.02594. Retrieved from https://arxiv.org/abs/2412.02594

[176] Zesong Jiang, Qing Zhang, Cheng Liu, Long Cheng, Huawei Li, and Xiaowei Li. 2024. Iicpilot: An intelligent integrated circuit backend design framework using open eda. arXiv:2407.12576. Retrieved from https://arxiv.org/abs/2407.12576

[177] Chia-Tung Ho and Haoxing Ren. 2024. Large language model (LLM) for standard cell layout design optimization. In *Proceedings of the 2024 IEEE LLM Aided Design Workshop (LAD'24)*. 1–6. DOI: http://doi.org/10.1109/LAD62341.2024.10691722

[178] Shan Huang, Jinhao Li, Zhen Yu, Jiancai Ye, Jiaming Xu, Ningyi Xu, and Guohao Dai. 2025. LLSM: LLM-enhanced logic synthesis model with EDA-guided CoT prompting, hybrid embedding and AIG-tailored acceleration. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC'25)*.

[179] Fahad Rahman Amik, Yousef Safari, Zhanguang Zhang, and Boris Vaisband. 2025. Graph-based timing prediction at early-stage RTL using large language model. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASPDAC'25)*. ACM, Tokyo, Japan, 393–400. DOI: http://doi.org/10.1145/3658617.3697742

[180] Reza Moravej, Saurabh Bodhe, Zhanguang Zhang, Didier Chetelat, Dimitrios Tsaras, Yingxue Zhang, Hui-Ling Zhen, Jianye Hao, and Mingxuan Yuan. 2024. The graph's apprentice: Teaching an LLM low level knowledge for circuit quality estimation. arXiv:2411.00843. Retrieved from https://arxiv.org/abs/2411.00843

[181] Ali Emre Oztas and Mahdi Jelodari. 2024. Agentic-HLS: An agentic reasoning based high-level synthesis system using large language models (AI for EDA workshop 2024). arXiv:2412.01604. Retrieved from https://arxiv.org/abs/2412.01604

[182] Zhuolun He and Bei Yu. 2024. Large language models for EDA: Future or mirage?. In *Proceedings of the 2024 International Symposium on Physical Design*. 65–66.

[183] Ruizhe Zhong, Xingbo Du, Shixiong Kai, Zhentao Tang, Siyuan Xu, Hui-Ling Zhen, Jianye Hao, Qiang Xu, Mingxuan Yuan, and Junchi Yan. 2023. Llm4eda: Emerging progress in large language models for electronic design automation. arXiv:2401.12224. Retrieved from https://arxiv.org/abs/2401.12224

[184] Jingyu Pan, Guanglei Zhou, Chen-Chia Chang, Isaac Jacobson, Jiang Hu, and Yiran Chen. 2025. A survey of research in large language models for electronic design automation. *ACM Transactions on Design Automation of Electronic Systems* (2025).

[185] Shadan Alsaqer, Sarah Alajmi, Imtiaz Ahmad, and Mohammad Alfailakawi. 2024. The potential of llms in hardware design. *Journal of Engineering Research* (2024).

[186] Kangwei Xu, Ruidi Qiu, Zhuorui Zhao, Grace Li Zhang, Ulf Schlichtmann, and Bing Li. 2024. LLM-aided efficient hardware design automation. arXiv:2410.18582. Retrieved from https://arxiv.org/abs/2410.18582

[187] Zhuolun He, Yuan Pu, Haoyuan Wu, Yuhan Qiu, Tairu Qiu, and Bei Yu. 2025. Large language models for EDA: From assistants to agents. *Foundations and Trends® in Electronic Design Automation* 14, 4 (2025), 295–314.

[188] Zeng Wang, Lilas Alrahis, Likhitha Mankali, Johann Knechtel, and Ozgur Sinanoglu. 2024. LLMs and the future of chip design: Unveiling security risks and building trust. arXiv:2405.07061. Retrieved from https://arxiv.org/abs/2405.07061

[189] Dipayan Saha, Shams Tarek, Katayoon Yahyaei, Sujan Kumar Saha, Jingbo Zhou, Mark Tehranipoor, and Farimah Farahmandi. 2024. Llm for soc security: A paradigm shift. *IEEE Access* 12 (2024), 155498–155521.

[190] Mohammad Akyash and Hadi Mardani Kamali. 2024. Evolutionary large language models for hardware security: A comparative survey. arXiv:2404.16651. Retrieved from https://arxiv.org/abs/2404.16651

[191] Lei Chen, Yiqi Chen, Zhufei Chu, Wenji Fang, Tsung-Yi Ho, Yu Huang, Sadaf Khan, Min Li, Xingquan Li, Yun Liang, et al. 2024. The dawn of ai-native eda: Promises and challenges of large circuit models. arXiv:2403.07257. Retrieved from https://arxiv.org/abs/2403.07257

[192] Khushboo Qayyum, Sallar Ahmadi-Pour, Chandan Kumar Jha, Muhammad Hassan, and Rolf Drechsler. 2024. LLMs for hardware verification: Frameworks, techniques, and future directions. In *Proceedings of the IEEE Asian Test Symposium* (2024), 1–6.

[193] Andrew B. Kahng. 2024. Solvers, engines, tools and flows: The next wave for AI/ML in physical design. In *Proceedings of the 2024 International Symposium on Physical Design*. 117–124.

[194] Synopsys Editorial Staff. 2019. The 6 Levels of Vehicle Autonomy Explained. (Apr 2019). Accessed: 2024-09-30. Retrieved from https://www.synopsys.com/blogs/chip-design/autonomous-driving-levels.html

[195] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* 4, 2 (2024), 100211.

[196] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)*. 2633–2650.

[197] Jean-Baptiste Truong, Pratyush Maini, Robert J. Walls, and Nicolas Papernot. 2021. Data-free model extraction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4771–4780.

[198] Nishant Subramani, Sasha Luccioni, Jesse Dodge, and Margaret Mitchell. 2023. Detecting personal information in training corpora: An analysis. In *Proceedings of the 3rd Workshop on Trustworthy Natural Language Processing (TrustNLP'23)*. 208–220.

[199] Zhaohan Xi, Tianyu Du, Changjiang Li, Ren Pang, Shouling Ji, Jinghui Chen, Fenglong Ma, and Ting Wang. 2024. Defending pre-trained language models as few-shot learners against backdoor attacks. *Advances in Neural Information Processing Systems* 36 (2024), 32748–32764.

[200] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys* 55, 12 (2023), 1–38.

[201] Rickard Stureborg, Dimitris Alikaniotis, and Yoshi Suhara. 2024. Large language models are inconsistent and biased evaluators. arXiv:2405.01724. Retrieved from https://arxiv.org/abs/2405.01724

[202] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards mitigating hallucination in large language models via self-reflection. arXiv:2310.06271. Retrieved from https://arxiv.org/abs/2310.06271

[203] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval augmentation reduces hallucination in conversation. arXiv:2104.07567. Retrieved from https://arxiv.org/abs/2104.07567

[204] Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. 2023. Can llms express their uncertainty? An empirical evaluation of confidence elicitation in llms. arXiv:2306.13063. Retrieved from https://arxiv.org/abs/2306.13063

[205] Zina Chkirbene, Ridha Hamila, Ala Gouissem, and Unal Devrim. 2024. Large language models (LLM) in industry: A survey of applications, challenges, and trends. In *Proceedings of the 2024 IEEE 21st International Conference on Smart Communities: Improving Quality of Life using AI, Robotics and IoT (HONET'24)*. IEEE, 229–234.

[206] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. 2024. GPTs are GPTs: Labor market impact potential of LLMs. *Science* 384, 6702 (2024), 1306–1308.

[207] Julian Senoner, Simon Schallmoser, Bernhard Kratzwald, Stefan Feuerriegel, and Torbjørn Netland. 2024. Explainable AI improves task performance in human–AI collaboration. *Scientific Reports* 14, 1 (2024), 31150.

[208] Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. 2024. Will we run out of data? Limits of LLM scaling based on human-generated data. arXiv:2211.04325. Retrieved from https://arxiv.org/abs/2211.04325

[209] Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang. 2024. On llms-driven synthetic data generation, curation, and evaluation: A survey. arXiv:2406.15126. Retrieved from https://arxiv.org/abs/2406.15126

[210] Bosheng Ding, Chengwei Qin, Ruochen Zhao, Tianze Luo, Xinze Li, Guizhen Chen, Wenhan Xia, Junjie Hu, Luu Anh Tuan, and Shafiq Joty. 2024. Data augmentation using llms: Data perspectives, learning paradigms and challenges. In *Findings of the Association for Computational Linguistics ACL 2024*. 1679–1705.

[211] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. arXiv:2212.10560. Retrieved from https://arxiv.org/abs/2212.10560

Received 4 October 2024; revised 17 February 2025; accepted 1 May 2025